

PATENT ABSTRACTS OF JAPAN

10/825,622

(11)Publication number : 11-122116

(43)Date of publication of application : 30.04.1999

(51)Int.Cl.

H03M 7/40
 G06F 17/14
 H03M 7/30
 H04N 1/41
 H04N 7/30

(21)Application number : 10-169417

(71)Applicant : CANON INF SYST RES AUSTRALIA
 PTY LTD
 CANON INC

(22)Date of filing : 30.04.1998

(72)Inventor : PROKOP TOMASZ THOMAS
 ELBOURNE TREVOR ROBERT
 PULVER MARK

(30)Priority

Priority number : 97 6486 Priority date : 30.04.1997 Priority country : AU
 97 6484 30.04.1997

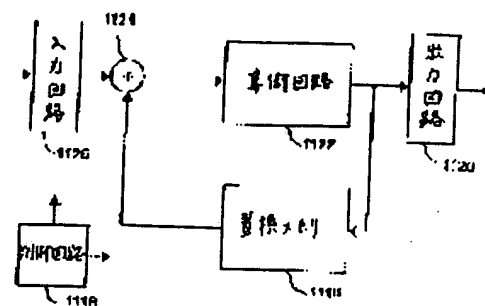
AU

(54) DEVICE AND METHOD FOR COMPRESSION

(57)Abstract:

PROBLEM TO BE SOLVED: To shorten the time needed for operation and to improve the performance of DCT(discrete cosine transformation) or reverse DCT by equipping a DCT device with a transposed matrix storage means and an arithmetic circuit consisting of a combinational circuit for performing DCT operation without using any clocked storage means.

SOLUTION: The substitute memory 1118 of the DCT transformation part transform column type data into row type data so as to implement 2nd pass of two-dimensional discrete cosine transformation. Data from an input circuit 1126 and the substitute memory 1118 are multiplexed by a multiplexer 1124 and sent to a mathematical circuit 1122. The result of the



mathematical circuit 1122 is sent to an output circuit 1120 after the 2nd pass ends. A control circuit 1116 controls a stream of data in the DCT transforming device. The mathematical circuit 1122 is the combinational circuit which does not have a storage location where an intermediate result is stored.

LEGAL STATUS

[Date of request for examination] 26.04.2005

[Date of sending the examiner's decision of rejection]

[Kind of final disposal of application other than the examiner's decision of rejection or application converted registration]

[Date of final disposal for application]

[Patent number]

[Date of registration]

[Number of appeal against examiner's decision of rejection]

[Date of requesting appeal against examiner's decision of rejection]

[Date of extinction of right]

Copyright (C); 1998,2003 Japan Patent Office

特開平11-122116

(43) 公開日 平成11年(1999) 4月30日

(51) Int.Cl.⁶

識別記号

F I

H 0 3 M 7/40

H 0 3 M 7/40

G 0 6 F 17/14

7/30

A

H 0 3 M 7/30

H 0 4 N 1/41

B

H 0 4 N 1/41

G 0 6 F 15/332

S

7/30

H 0 4 N 7/133

Z

審査請求 未請求 請求項の数25 書面 外国語出願 (全748頁)

(21) 出願番号 特願平10-169417

(22) 出願日 平成10年(1998) 4月30日

(31) 優先権主張番号 P O 6 4 8 6

(32) 優先日 1997年4月30日

(33) 優先権主張国 オーストラリア (AU)

(31) 優先権主張番号 P O 6 4 8 4

(32) 優先日 1997年4月30日

(33) 優先権主張国 オーストラリア (AU)

(71) 出願人 591146745

キャノン インフォメーション システム
ズ リサーチ オーストラリア プロプラ
イエタリー リミテッドCANON INFORMATION S
YSTEMS RESEARCH AUS
TRALIA PTY LTDオーストラリア国 2113 ニュー サウス
ウェールズ州, ノース ライド, ト
ーマス ホルト ドライブ 1

(74) 代理人 弁理士 大塚 康徳 (外2名)

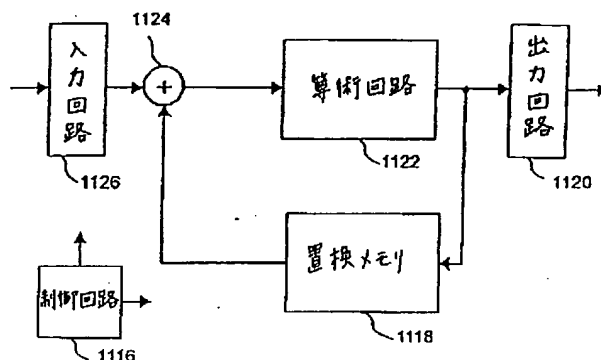
最終頁に続く

(54) 【発明の名称】 圧縮装置及びその方法

(57) 【要約】

【課題】 高速動作可能なDCT/逆DCT装置ならびに方法を提供すること。

【解決手段】 転置行列メモリ(1118)に接続された演算回路(1122)と、DCT演算を行う、中間のクロック記憶回路を有さない組み合わせ回路(1138)を有するDCT装置(1118-1126)であって、組み合わせ回路はDCTを行うためのシーケンシャルに配置された所定数のステージ(1158-1164)を有し、またDCT装置はさらに入力データと転置行列メモリ(1118)の出力データを多重化する多重化器(1124)を有していても良い。また、DCT装置の動作を制御する制御回路(1116)を有することが好ましい。



【特許請求の範囲】

【請求項1】 可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードにより符号化され、複数の固定長の非符号化フィールドを有する複数のデータブロックの復号化装置であって、
複数の固定長非符号化フィールドを除去し、前記可変長非符号化ビットフィールドと前記可変長の非符号化ビットフィールドでインターリーブされた前記複数の可変長コードと、複数のデータブロック中の前記複数の固定長非符号化フィールドの位置を示す複数の位置信号とを出力する前処理部と、
前記固定長非符号化フィールドの間に入力される可変長符号化データの復号化データが、前記固定長非符号化フィールドに対応する前記位置信号の間に復号化装置から出力されるように、前記位置信号を復号化されるデータと同期させて外部装置へ送出する手段とを含むことを特徴とする復号化装置。

【請求項2】 第1のバレルシフトセットと第1レジスタを有し、前記可変長の非符号化ビットフィールドでインターリーブされた前記複数の可変長コードを処理する第1処理部と、
第2のバレルシフトセットと第2レジスタを有し、複数のデータブロック中の前記複数の固定長非符号化フィールドの位置を示す複数の位置信号を処理する第2処理部とを更に有し、前記第1および第2処理部が同一であり、前記第1、第2バレルシフトセットの出力と前記第1、第2処理部が同じ制御信号を受信する請求項1記載の復号化装置。

【請求項3】 前記固定長非符号化フィールドの位置を示す位置信号を処理する前記第2処理部の出力が、データレジスタに保管されたデータから復号化時に除去される非符号化可変長フィールドのサイズ決定に用いられることを特徴とする請求項2記載の復号化装置。

【請求項4】 前記前処理部が、複数の固定長非符号化フィールドを除去し、前記可変長非符号化ビットフィールドと前記可変長の非符号化ビットフィールドでインターリーブされた前記複数の可変長コードを、複数の固定長ビットフィールドからなる複数の固定長コードとして、かつ一つの前記固定長ビットフィールドが、前記前処理フィールドでパス又は除去されたこと、前記前処理フィールドでパスされたことのいずれかを示すタグを有し、かつ前記タグは固定長の非符号化フィールドを示すマーカーの前又は後ろに存在するように出力することを特徴とする請求項1の復号化装置。

【請求項5】 前記データブロックがハフマン符号化されていることを特徴とする請求項1の復号化装置。

【請求項6】 可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードにより符号化され、複数の固定長の非符号化フィールドを有する複数のデータブロックの復号化方法であって、

複数の固定長非符号化フィールドを除去し、前記可変長非符号化ビットフィールドと前記可変長の非符号化ビットフィールドでインターリーブされた前記複数の可変長コードと、複数のデータブロック中の前記複数の固定長非符号化フィールドの位置を示す複数の位置信号とを出力する前処理ステップと、

前記固定長非符号化フィールドの間に入力される可変長符号化データの復号化データが、前記固定長非符号化フィールドに対応する前記位置信号の間に復号化装置から出力されるように、前記位置信号を復号化されるデータと同期させて外部装置へ送出するステップとを含むことを特徴とする復号化方法。

【請求項7】 第1のバレルシフトセットと第1レジスタを有する第1処理部を用いて、前記可変長の非符号化ビットフィールドでインターリーブされた前記複数の可変長コードを処理するステップと、
第2のバレルシフトセットと第2レジスタを有する第2処理部を用いて、複数のデータブロック中の前記複数の固定長非符号化フィールドの位置を示す複数の位置信号を処理するステップを更に有し、
前記第1および第2処理部が同一であり、前記第1、第2バレルシフトセットの出力と前記第1、第2処理部が同じ制御信号を受信する請求項6記載の復号化方法。

【請求項8】 前記固定長非符号化フィールドの位置を示す位置信号を処理する前記第2処理部の出力に応じて、データレジスタに保管されたデータから復号化時に除去される非符号化可変長フィールドのサイズを決定するステップを更に有することを特徴とする請求項7記載の復号化方法。

【請求項9】 前記前処理部ステップにおいて、複数の固定長非符号化フィールドを除去し、前記可変長非符号化ビットフィールドと前記可変長の非符号化ビットフィールドでインターリーブされた前記複数の可変長コードを、複数の固定長ビットフィールドからなる複数の固定長コードとして、かつ一つの前記固定長ビットフィールドが、前記前処理フィールドでパス又は除去されたこと、前記前処理フィールドでパスされたことのいずれかを示すタグを有し、かつ前記タグは固定長の非符号化フィールドを示すマーカーの前又は後ろに存在するように出力することを特徴とする請求項6記載の復号化方法。

【請求項10】 前記データブロックがハフマン符号化されていることを特徴とする請求項6記載の復号化方法。

【請求項11】 離散コサイン変換(DCT)装置であって、転置行列記憶手段と、該転置行列記憶手段と相互接続され、clocked storage手段を用いずにDCT演算を行うための組合せ回路からなる演算回路を有するDCT装置。

【請求項12】 前記組合せ回路がDCT演算を行うための所定数のステージを有し、該ステージがシーケンシ

ャルに配置されている請求項11のDCT装置。

【請求項13】 前記DCT装置に入力されるデータと、前記転置行列記憶手段の出力とを多重化する多重化手段を有する請求項11記載のDCT装置。

【請求項14】 前記DCT装置の動作を制御する制御手段を有する請求項11記載のDCT装置。

【請求項15】 逆離散コサイン変換(DCT)装置であって、
転置行列記憶手段と、
該転置行列記憶手段と相互接続され、clocked storage手段を用いずにDCT演算を行うための組合せ回路を主構成要素とする演算回路を有する逆DCT装置。

【請求項16】 前記組合せ回路が逆DCT演算を行うための所定数のステージを有し、該ステージがシーケンシャルに配置されている請求項15の逆DCT装置。

【請求項17】 前記逆DCT装置に入力されるデータと、前記転置行列記憶手段の出力とを多重化する多重化手段を有する請求項15記載の逆DCT装置。

【請求項18】 前記逆DCT装置の動作を制御する制御手段を有する請求項15記載の逆DCT装置。

【請求項19】 データの離散コサイン変換(DCT)方法であって、
DCT演算をclocked storage手段無しで行う組合せ回路を主要構成とする演算回路を用い、入力データをその第1の方向に合わせてDCT演算するステップと、
DCTされた入力データを前記第1の方向に合わせて前記組合せ回路と相互接続された転置行列記憶手段に記憶するステップと、
前記演算回路を用い、前記転置行列記憶手段に記憶されたデータをその第2の方向に合わせてDCT演算して変換データを得るステップとを有する方法。

【請求項20】 前記DCTがシーケンシャルに配置された所定数のステージで演算される請求項19記載のDCT方法。

【請求項21】 入力されるデータと前記転置行列記憶手段の出力とを多重化するステップを有する請求項19記載のDCT方法。

【請求項22】 逆離散コサイン変換(DCT)方法であって、
逆DCT演算をclocked storage手段無しで行う組合せ回路を主要構成とする演算回路を用い、入力係数をその第1の方向に合わせてDCT演算するステップと、
DCTされた入力係数を前記第1の方向に合わせて前記組合せ回路と相互接続された転置行列記憶手段に記憶するステップと、
前記演算回路を用い、前記転置行列記憶手段に記憶された係数をその第2の方向に合わせて逆DCT演算して逆

変換データを得るステップとを有する方法。

【請求項23】 前記逆DCTがシーケンシャルに配置された所定数のステージで演算される請求項22記載の逆DCT方法。

【請求項24】 入力されるデータと前記転置行列記憶手段の出力とを多重化するステップを有する請求項22記載の逆DCT方法。

【請求項25】 請求項6～10および請求項19～24のいずれかに記載の方法をコンピュータが実行可能なプログラムコードとして記憶する記憶媒体。

【発明の詳細な説明】

【0001】

【発明の属する技術分野】本発明は符号化されていない0またはそれ以上の可変長ビットフィールドを挿入された可変長コードを復号化する復号化器に関し、コードのいくつかは変化させない復号化器に関する。本発明はまた、パイプラインまたは記憶手段を持たないデータパスを用い、かつ高速で動作可能な離散コサイン変換(DCT)装置に関する。

【0002】

【従来の技術】一般に、大きな量のデータは電送、記憶、読み出しおよびハフマン符号化のような可変長符号化のいくつかのステージ手段での使用を含むさまざまな理由により圧縮されまた伸長される。ハフマン符号化はD. A. ハフマンによる論文、「最小冗長コードの構築方法("A Method for the Construction of Minimum Redundancy Codes" Proc. IRE, 40:1098, 1952)」により最初に開示された。多くの場合、符号化ビット列中の可変長符号は不連続であり、他の非符号化ビットフィールドが挿入されている。このビットフィールドは制御および/またはフォーマット情報を表し、かつ/またはマーカーヘッダ、マーカーコード、スタンプバイト、パディングビットおよび含まれる付加ビット、たとえばJPEG符号化データなどを含む、符号化データに対する追加事項を供給する。

【0003】可変長符号化においては、統計的に頻度の高い入力コードが頻度の低いデータよりも短い符号を割り当てられるように、入力データの発生確からしさに基づいて異なる入力データに異なる長さの符号を割り当てる。頻度の低い入力コードは長いコードを割り当てられる。コードの割り当ては統計的もしくは適応的のいずれかによりなされる。統計的割り当ての場合、どのデータブロックが処理されるかに関わらず一定のデータには同じ出力コードが与えられる。適応的割り当ての場合、出力コードは特定の入力ブロックまたはデータブロックのセットの統計的分析および予想されるブロック間(またはブロックセット間)での変化に基づき割り当てられる。

【0004】高速な可変長復号化が必要となった場合、

重大な問題が起きる。問題は特にJ P E G標準のような、符号化データ列が符号化データを挿入（インターリーブ）された符号化されていない可変長のビットフィールドを含む場合におきる。そのような可変長符号化データの高速復号化における大きな困難は、J P E G標準のように特定の非符号化ビットフィールドの長さが引き続く（符号化された）データの復号化が完全に終了した後でないと判別できない場合に発生する。次の符号化データの開始位置が、後ろのコードの復号化が完全に終わった後でないとわからないため、一般的に直接パイプライン処理を復号化器とともに用いることができない。

【0005】現存する解決法は、多くの用途に対して遅すぎるが、一つの入力データの復号化に数ステップ（クロックサイクル）を必要とするか、繰り返しユニット（iterative units）を用いて一つより多いシンボルを1ステージ（クロックサイクル）で擬似的に同時復号化するかである。しかし、更なる復号化ブロックの追加はしばしばそのような復号化器を経済的に釣り合わなくし、さらに必要十分な速度を得られなくする。これは次の復号化器の処理開始が依然として次の入力データの先頭を決定する、前に位置する復号化器の処理結果に依存するため、複数の復号化器が完全な並列動作をしないからである。その結果、1ステージ（クロックサイクル）で複数のシンボルを復号化したとしても、そのステージ（クロック期間）は相対的に長く、全体の復号化器としては多くの用途において遅すぎることになる。

【0006】よって、従来の復号化器の問題点を1つかそれ以上解決した、可変長非符号化ビットフィールドでインターリーブされた可変長コードの復号化器に対する要求は明らかに存在する。具体的には、図77に示された離散コサイン変換（DCT）装置は 8×8 画素のブロックの完全二次元（2-D）変換を 8×8 画素ブロックの行にまず1-DDCTを行うことで実現する。そして、別の1-DDCTを 8×8 画素ブロックの列に対して行う。このような装置は具体的には入力回路1096、演算回路1104、制御回路1098、転置行列メモリ回路1090及び出力回路1092から構成される。

【0007】入力回路1096は 8×8 ブロックから8ビットの画素を受け付ける。入力回路1096は中間多重化器1100、1102によって演算回路1104に接続される。演算回路1104は 8×8 ブロックの行または列全体のいずれかに算術操作を施す。制御回路1098は他のすべての回路を制御し、DCTアルゴリズムを実現する。演算回路の出力は転置行列メモリ1090、レジスタ1095および出力回路1092に接続される。転置行列メモリは次の多重化器1102に出力を供給する多重化器1100に接続されている。多重化器1102はまたレジスタ1094から入力を受信す

る。転置行列メモリ1090は 8×8 ブロックのデータを行に受け付け、列にデータを生成する。出力回路1092は 8×8 の画素データブロックになされるDCTの係数を供給する。

【0008】典型的なDCT装置において、演算回路がもともと複雑であるため、基本的には演算回路1104の速度が全体の速度を決定付ける。図77における演算回路1104は、演算処理を図78を参照して以下に説明されるいくつかのステージに分割して構成されている。これらのステージ1114、1148、1152、1156は加算器や乗算器などの共通利用されるものの集まりによって一つの回路で実現することができる。しかし、そのような回路1104は、共通に使われる一つの回路で複数のステージを構成しているため、最適化されたものに比べて遅いという欠点がある。これは中間結果の記憶に用いる記憶手段を含む。そのような回路のクロックサイクルとして割り当てられる時間は、回路の中の最も遅いステージの時間と同じかそれ以上でなければならない、全体としての時間は全部のステージの合計よりも長くなる可能性があるからである。

【0009】図78は図77に示した装置における、4ステージのDCTの一部としての典型的な演算データパスを示している。図は現実の構成を反映しているわけではないが、機能は反映している。4つのステージ1144、1148、1152、1156のそれぞれは一つの、再構成可能な回路で構成されている。1サイクル毎に1-DDCTである各4つの演算ステージ1144、1148、1152、1156が再構成される。この回路において、4つのステージ1144、1148、1152、1156のそれぞれは共通に使用されるもの（加算器および乗算器）の集まりを使用しており、ハードウェアを最小化している。

【0010】しかしながらこの回路の欠点は、最適化されたものに比べて遅いということである。4つのステージ1144、1148、1152、1156のそれぞれは同一の加算器および乗算器の集まりで構成されている。クロック周期は最も遅いステージで決まり、この例ではブロック1144における 20 ns である。入力及び出力多重化器1146および1154の遅延（各 2 ns ）およびフリップフロップ1150の遅延（ 3 ns ）を加えると、合計時間は 27 ns である。よって、このDCT要素は 27 ns で動作することが可能である。

【0011】パイプライン化されたDCT要素もまたよく知られている。この構成の問題点は構成に多量のハードウェアを必要とすることである。本発明では同じ性能すなわち処理速度を提供はしないが、非常によい性能対大きさの妥協点を提供する。さらに、現状のDCT要素の大半よりもよい速度的な利点を提供する。よって、従来の技術が有する1つまたはそれ以上の課題を解決できる、改良されたDCT／逆DCT方法及び装置に対する

要求は明確である。特に、DCT／逆DCT装置において必要な結果を計算する中心的な演算回路に要する時間を短縮でき、DCTまたは逆DCT全体の性能を向上する方法および装置の必要性は明確である。

【0012】

【課題を解決するための手段】本発明の第1は、可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードにより符号化されたデータと、符号化されない固定長の非符号化フィールドを有する複数のデータブロックの復号化装置であって、複数の固定長非符号化フィールドを除去し、可変長非符号化ビットフィールドと可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードと、複数のデータブロック中の複数の固定長非符号化フィールドの位置を示す複数の位置信号とを出力する前処理部と、固定長非符号化フィールドの間に入力される可変長符号化データの復号化データが、固定長非符号化フィールドに対応する位置信号の間に復号化装置から出力されるように、位置信号を復号化されるデータと同期させて復号化装置の出力へ受け渡しする受け渡し手段とを含む復号化装置である。

【0013】また、好ましくは、第1のバレルシフトセットと第1レジスタを有し、可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードを処理する第1処理部と、第2のバレルシフトセットと第2レジスタを有し、複数のデータブロック中の複数の固定長非符号化フィールドの位置を示す複数の位置信号を処理する第2処理部とを更に有し、第1および第2処理部が同一であり、第1、第2バレルシフトセットの出力と第1、第2処理部が同じ制御信号を受信する復号化装置である。

【0014】好ましい別の構成としては、固定長非符号化フィールドの位置を示す位置信号を処理する第2処理部の出力が、データレジスタに保管されたデータから復号化時に除去される非符号化可変長フィールドのサイズ決定に用いられる復号化装置である。さらに、別の好ましい構成としては、前処理部が、複数の固定長非符号化フィールドを除去し、可変長非符号化ビットフィールドと可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードを、複数の固定長ビットフィールドからなる複数の固定長コードとして、かつ一つの固定長ビットフィールドが、前処理フィールドでパス又は除去されたこと、前処理フィールドでパスされたことのいずれかを示すタグを有し、かつタグは固定長の非符号化フィールドを示すマーカーの前又は後ろに存在するように出力する復号化装置である。

【0015】データブロックがハフマン符号化されていることがさらに好ましい。また、本発明の第2は、可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードにより符号化されたデータと、符号

化されない固定長の非符号化フィールドを有する複数のデータブロックの復号化方法であって、複数の固定長非符号化フィールドを除去し、可変長非符号化ビットフィールドと可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードと、複数のデータブロック中の複数の固定長非符号化フィールドの位置を示す複数の位置信号とを出力する前処理ステップと、固定長非符号化フィールドの間に入力される可変長符号化データの復号化データが、固定長非符号化フィールドに対応する位置信号の間に復号化装置から出力されるように、位置信号を復号化されるデータと同期させて復号化装置の出力へ受け渡しする受け渡しステップとを含む復号化方法である。

【0016】好ましくは、第1のバレルシフトセットと第1レジスタを有する第1処理部を用いて、可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードを処理するステップと、第2のバレルシフトセットと第2レジスタを有する第2処理部を用いて、複数のデータブロック中の複数の固定長非符号化フィールドの位置を示す複数の位置信号を処理するステップを更に有し、第1および第2処理部が同一であり、第1、第2バレルシフトセットの出力と第1、第2処理部が同じ制御信号を受信する復号化方法である。

【0017】さらには、固定長非符号化フィールドの位置を示す位置信号を処理する第2処理部の出力に応じて、データレジスタに保管されたデータから復号化時に除去される非符号化可変長フィールドのサイズを決定するステップを更に有する復号化方法である。好ましくは、前処理部ステップにおいて、複数の固定長非符号化フィールドを除去し、可変長非符号化ビットフィールドと可変長の非符号化ビットフィールドでインターリーブされた複数の可変長コードを、複数の固定長ビットフィールドからなる複数の固定長コードとして、かつ一つの固定長ビットフィールドが、前処理フィールドでパス又は除去されたこと、前処理フィールドでパスされたことのいずれかを示すタグを有し、かつタグは固定長の非符号化フィールドを示すマーカーの前又は後ろに存在するように出力する復号化方法である。

【0018】また、データブロックがハフマン符号化されていることが好ましい。以下の詳細な説明においては、他の説明はもとより、特に図82～91およびそれに関係する説明に注意されたい。本発明の第3は、離散コサイン変換(DCT)装置であって、転置行列記憶手段と、転置行列記憶手段と相互接続され、clocked storage手段を用いずにDCT演算を行うための組合せ回路からなる演算回路を有するDCT装置である。

【0019】好ましくは、組合せ回路がDCT演算を行うための所定数のステージを有し、ステージがシーケンシャルに配置DCT装置である。また好ましくは、DC

T装置に入力されるデータと、転置行列記憶手段の出力とを多重化する多重化手段を有するDCT装置である。またDCT装置の動作を制御する制御手段を有するDCT装置である。

【0020】本発明の第4は、逆離散コサイン変換(DCT)装置であって、転置行列記憶手段と、転置行列記憶手段と相互接続され、clocked storage手段を用いずにDCT演算を行うための組合せ回路を主構成要素とする演算回路を有する逆DCT装置である。

【0021】本発明の第5は、データの離散コサイン変換(DCT)方法であって、DCT演算をclocked storage手段無しで行う組合せ回路を主要構成とする演算回路を用い、入力データをその第1の方向に合わせてDCT演算するステップと、DCTされた入力データを第1の方向に合わせて組合せ回路と相互接続された転置行列記憶手段に記憶するステップと、演算回路を用い、転置行列記憶手段に記憶されたデータをその第2の方向に合わせてDCT演算して変換データを得るステップとを有する方法である。

【0022】好ましくは、DCTがシーケンシャルに配置された所定数のステージで演算されるDCT方法であり、また入力されるデータと転置行列記憶手段の出力とを多重化するステップを有してもよい。本発明の第6は、逆離散コサイン変換(DCT)方法であって、逆DCT演算をclocked storage手段無しで行う組合せ回路を主要構成とする演算回路を用い、入力係数をその第1の方向に合わせてDCT演算するステップと、DCTされた入力係数を第1の方向に合わせて組合せ回路と相互接続された転置行列記憶手段に記憶するステップと、演算回路を用い、転置行列記憶手段に記憶された係数をその第2の方向に合わせて逆DCT演算して逆変換データを得るステップとを有する方法である。

【0023】以下の詳細な説明においては、他の説明はもとより、特に図79、80および81およびそれに関する説明に注意されたい。

【0024】

【発明の実施の形態】

「目次」

- 1. 0 図面の簡単な説明
- 2. 0 テーブルリスト
- 3. 0 好適な及び他の実施例
- 3. 1 複数のストリームアーキテクチャの概要
- 3. 2 ホスト/コプロセッサのキューイング
- 3. 3 コプロセッサのレジスタ説明
- 3. 4 複数のストリームのフォーマット
- 3. 5 現アクティブストリームの判定
- 3. 6 現アクティブストリームのフェッチ命令
- 3. 7 命令のデコード及び実行
- 3. 8 命令コントローラのレジスタの更新

- 3. 9 レジスタアクセスセマフォの意味論
- 3. 10 命令コントローラ
- 3. 11 ローカルレジスタファイルモジュールの説明
- 3. 12 レジスタのリード・ライト処理
- 3. 13 メモリエリアのリード/ライト処理
- 3. 14 Cバス構造
- 3. 15 コプロセッサのデータタイプとデータ操作
- 3. 16 データ正規化処理
- 3. 17 アクセラレータカードの画像処理
 - 3. 17. 1 合成
 - 3. 17. 2 色空間変換命令
 - a. 単一出力カラー空間(SOGCS)変換モード
 - b. 複数出力から一空間モード
 - 3. 17. 3 LPEG符号化/復号化
 - a. 符号化
 - b. 復号化
 - 3. 17. 4 テーブル索引
 - 3. 17. 5 データ符号化命令
 - 3. 17. 6 高速DCT装置
 - 3. 17. 7 ハフマン復号
 - 3. 17. 8 イメージ変換命令
 - 3. 17. 9 コンボルジョン命令
 - 3. 17. 10 マトリクス乗算
 - 3. 17. 11 階調(ハーフトーン)
 - 3. 17. 12 階層イメージフォーマット伸長
 - 3. 17. 13 メモリコピー命令
 - a. 汎用データ移動命令
 - b. ローカルDMA命令
 - 3. 17. 14 フロー制御命令
- 3. 18 アクセラレータカードのモジュール
 - 3. 18. 1 ピクセルオーガナイザ
 - 3. 18. 2 MUVバッファ
 - 3. 18. 3 結果オーガナイザ
 - 3. 18. 4 オペランドオーガナイザB, C
 - 3. 18. 5 メインデータバスユニット
 - 3. 18. 6 データキャッシュコントローラとキャッシュ
 - a. ノーマルキャッシュモード
 - b. 単一出力一般カラー空間変換モード
 - c. 複数出力一般カラー空間変換モード
 - d. JPEG符号化モード
 - e. 低速JPEG復号モード
 - f. マトリクス乗算モード
 - g. ディスエーブルモード
 - h. 無効化モード
 - 3. 18. 7 入力インターフェーススイッチ
 - 3. 18. 8 ローカルメモリコントローラ
 - 3. 18. 9 その他のモード
 - 3. 18. 10 外部インターフェースコントローラ
 - 3. 18. 11 周辺インターフェースコントローラ

テーブル索引

テーブル1:	レジスタの説明
テーブル2:	オペコードの説明
テーブル3:	オペランドタイプ
テーブル4:	オペランド説明
テーブル5:	モジュールセットアップ順序
テーブル6:	Cバス信号の定義
テーブル7:	Cバスのトランザクションタイプ
テーブル8:	データ操作レジスタフォーマット
テーブル9:	希望データタイプ
テーブル10:	シンボル説明
テーブル11:	合成処理
テーブル12:	SOGCSモード用アドレス合成
テーブル12A:	色空間変換用命令符号化
テーブル13:	色変換命令用のマイナーオペコード符号化
テーブル14:	データキャッシュに記憶されたハフマン及び量子化テーブル
テーブル15:	フェッチアドレス
テーブル16:	ハフマン符号化用テーブル
テーブル17:	ハフマン及び量子化テーブル用バンクアドレス
テーブル18:	命令ワードマイナーオペコードフィールド
テーブル19:	命令ワードマイナーオペコードフィールド
テーブル20:	命令オペランドー結果ワード
テーブル21:	命令ワード
テーブル22:	命令オペランドー結果ワード
テーブル23:	命令ワード
テーブル24:	命令オペランドー結果ワード
テーブル25:	命令ワードマイナーオペコードフィールド
テーブル26:	命令ワードマイナーオペコードフィールド
テーブル27:	分数テーブル

【好適ならびに他の実施例の説明】好適な実施例では、ハードウェアアクセラレータによる2つの独立命令ストリームの利用によってハードウェアラスティングを行うことで大きな利点が得られている。従って、第一の命令ストリームが現ページの印刷準備をしている間に、次の命令ストリームが次ページの印刷準備をすることができる。ハードウェア資源は、ハードウェアアクセラレータが出力装置以上の速度で動作可能である場合に特に効率的に利用することができる。

【0025】好適な実施例では、2命令ストリームを用いる構成を示す。しかし、2以上の命令ストリームを用いる構成も可能であり、ハードウェアトレードオフを鑑みてもより多くのストリームを用いることによる利点が得られる。2つのストリームを用いることで、ラスタ画

像コプロセッサのハードウェア資源は、出力装置に応じて現ページ、バンド、ストリップなどを印刷装置に転送している間にも、続くページ、バンド、ストリップなどの準備に常に関与することができる。

3. 1 複数ストリームアーキテクチャの一般構成

図1は、好適な実施例を含むコンピュータハードウェア構成201を模式的に示した図である。構成201には、ブリッジ204を介してホスト記憶メモリ203に接続されたホストCPU202から成る標準ホストコンピュータシステムが含まれている。ホストコンピュータシステムには、オペレーティングシステムプログラム、アプリケーション、情報ディスプレイなどの一般のコンピュータシステム機能が備わっており、ホストコンピュータシステムはPCIバスインタフェース207を介して標準PCIバス206に接続されている。なお、PCI標準は良く知られた業界標準であり、市販のほとんどのコンピュータシステム、特にマイクロソフトウィンドウズ(商標)オペレーティングシステムを搭載しているシステムには、PCIバス206が備わっている。PCIバス206を用いることにより、PCIバスインタフェース210、他のデバイス211、ローカルメモリ212などを更に含む1つ或は複数のPCIカード(例えば209)を構成201に付加して利用することが容易になる。

【0026】好適な実施例では、ページ記述言語で表現されたグラフィックス処理を高速にするために、ラスタ画像アクセラレータカード220を備える。ラスタ画像アクセラレータカード(PCIバスインタフェース221を備える)は、他のPCIカード209などと同様にホストCPU202とは、緩やかに結合された共有メモリの形態で動作するように設計されている。なお、必要であれば、画像アクセラレータカード220を更にホストコンピュータシステムに付加することもできる。ラスタ画像アクセラレータカードは、ラスタ画像処理動作における複雑かつ多量の動作処理を高速化するためのものであり、これらの動作としては、

- (a) 合成
- (b) 一般化色空間変換
- (c) JPEG符号化/復号
- (d) ハフマン、ランレングス、予測符号化/復号
- (e) 階層的画像(商標)復号
- (f) 一般化アフィン画像変換
- (g) 小カーネル畳込演算(コンボルーション)
- (h) 行列演算
- (i) ハーフトーン処理
- (j) 一括算術/メモリコピー演算

ラスタ画像アクセラレータカード220は更にラスタ画像コプロセッサ224に接続されたローカルメモリ223を備え、ラスタ画像コプロセッサ224はホストCPU202からの命令に基づいてラスタ画像アクセラレー

タカード220を起動する。ここで、コプロセッサ224は特定用途向けLSI（ASIC）であることが望ましい。また、ラスタ画像コプロセッサ224は、必要少なくとも1つのプリンタデバイス226を周辺インタフェース225を介して制御する能力を有する。更に、画像アクセラレータカード220は、スキャナなどの入力／出力デバイスを制御することも可能である。あわせて、アクセラレータカード220にはラスタ画像コプロセッサ224に接続された一般外部インタフェース227が備えられており、モニタリングやテストを行うこともできる。。

【0027】実行モードでは、ホストCPU202がPCIバス206を介して一連の命令やデータを送信し、ラスタ画像コプロセッサ224で画像の生成処理を行う。送信されたデータはローカルメモリ223のみならずラスタ画像コプロセッサ224中のキャッシュ230、あるいはコプロセッサ224中のレジスタ229に蓄えられる。

【0028】図2は、ラスタ画像コプロセッサ224をより詳細に示した図である。コプロセッサ224は、前記の処理を高速化するためのものであり、命令制御部235の制御下にある複数の部位から構成される。コプロセッサが外界と通信するために、図1のローカルメモリ223と通信するためのローカルメモリ制御部236を具備している。周辺インタフェース制御部237は、プリンタデバイスとの通信に利用されるもので、セントロニクスインタフェース標準フォーマットや他のビデオインタフェースフォーマットなどの標準フォーマットを利用する。周辺インタフェース制御部237はローカルメモリ制御部236と内部接続されている。ローカルメモリ制御部236と外部インタフェース制御部238とは入力インタフェーススイッチ252を介して接続されており、入力インタフェーススイッチ252は命令制御部235と接続されている。入力インタフェーススイッチ252はまたピクセルオーガナイザ246とデータキャッシュ制御部240に接続されている。入力インタフェーススイッチ252は、外部インタフェース制御部237とローカルメモリ制御部236からのデータをスイッチして命令制御部235、あるいはデータキャッシュ制御部240、ピクセルオーガナイザ246に転送するためのものである。

【0029】外部インタフェース制御部238は、図1中のPCIバス206と通信するためにラスタ画像コプロセッサ224中に具備されており、命令制御部235と接続されている。また、テスト診断を行ったり、クロック信号やグローバル信号を入力するために、命令制御部239に接続され、コプロセッサ224と協調して動作する他モジュール239が備わっている。

【0030】データキャッシュ230は、接続されているデータキャッシュ制御部240の制御下で動作する。

データキャッシュ230は種々の用途において用いられるが、コプロセッサ224において引き続き使用される確率の高い最近使用した値を蓄えるために主として用いられる。上述の高速化処理は、主としてJPEG符号化／復号器241やメインデータパス部242によって複数のデータストリームの処理が行われる。部位241、242は並列にピクセルオーガナイザ246と2つのオペランドオーガナイザ247、248に接続されている。部位241、242からの処理されたストリームは、結果オーガナイザ249に転送され、必要であれば処理や再フォーマット処理が行われる。なお、中間結果を記録しておきたいことも多いため、データキャッシュ230に加えて、ピクセルオーガナイザ246と結果オーガナイザ249との間にマルチユースト値（MUV）バッファ250を備えている。結果オーガナイザ249からの結果は、必要であれば外部インタフェース制御部238、ローカルメモリ制御部236、周辺インタフェース制御部237に出力される。

【0031】図2中の点線で示されているように、さらなる（第3の）データパス部243を、JPEG符号化／復号器241とメインデータパス部242といった他の二つのデータパスと「並列に」接続することも可能である。また、四あるいはそれ以上のデータパスを構成することも同様に可能である。なお、パスは「並列に」接続されているが、並列に動作するものではなく、一つのパスのみが一時に動作するものであることに注意されたい。

【0032】図2のASICの全体設計は以下のような考えに基づいてなされた。まず第1に、印刷ページでは小さな、或は一時的な画質劣化をも生じさせないことが必須である。映像信号では、このような小さな画質劣化が存在したとしても人間の目では感知されることはないが、印刷物では印刷ページに永久的に小さな画質劣化が残ってしまい、目立つようになることもあるからである。更に、プリンタに至るまでに遅延が生じると、ページがプリンタ内を移動している間に白い未印刷の部位がページ上にできてしまうことがあるため、見苦しいものとなる。そのため、高品質かつ高速に結果を提供することが必須となり、ソフトウェアを用いるアプローチよりもハードウェアの高速性に頼るアプローチの方が好ましい。

【0033】第2に、印刷処理を実行するのに必要なさまざまな動作ステップ（アルゴリズム）すべてをリストアップし、各ステップごとに対応するハードウェアを並べ上げると、全体のハードウェア量は膨大なものになり、非常に高価なものになってしまう。また、ハードウェアの動作スピードは、処理に必要なデータをフェッチしたり、あるいは処理で生成されたデータを転送するレートによって本質的に制限される。すなわち、動作スピードはインタフェースの帯域幅によって制約を受ける。

【0034】これに対して、全体のASICのデザインは、ハードウェアの全体量を模式的に表したときに、必要なハードウェアの種々の部位が（a）重複しており、（b）同時に実行されることはない、という驚くべき事実に基づいている。特に、この点はデータの処理をする前にデータを転送する際のオーバーヘッドにおいて顕著にみられる。

【0035】このような観点から、いくつかのステップを経て、ハードウェアのすべての部位をできるだけアクティブにしながら、ハードウェア量を低減することにした。第1のステップにおいて、画像操作では多くの場合同一の基本的種類の繰返し演算が必要であることを認識した。従って、データがストリーム状に入力されると、特定の処理を行うように処理部を構成して長いデータストリームを処理し、その後次に必要な処理タイプに合うように処理部を再構成する。データストリームがかなり長いと、再構成に要する時間は全体の処理時間と比較して無視できるほど短くなるため、スループットが向上することになる。

【0036】また、複数のデータ処理パスを設けると、他のパスを使用している間に一つのパスを再構成することで、再構成に要する時間の無駄を省くこともできる。すなわち、メインデータパス部242がより汎用的な処理を実行している間に、他のデータパスにおいて部位241のようなJPEG符号化／復号、あるいは追加部位243がある場合にはエントロピー符号化やハフマン符号化などのより特化した処理を行うことができる。

【0037】更に、処理を進めている間に、処理部位へのデータのフェッチや転送を行うこともできる。また、種々の種別のデータを標準化、統一することにより、更に高速化を図ることができるとともに、ハードウェア資源も有効に利用することができる。従って、データのフェッチや転送に関わる全体のオーバーヘッドを低減することができる。

【0038】ここで重要なことは、コプロセッサ224がホストCPU202（図1）の制御の下で実行されることである。この点で、命令制御部235が、コプロセッサ224全体の制御を統括する。命令制御部235は、CBus（Cバス）と呼ばれる制御バス231によってコプロセッサ224を動作させる。CBus231はそれぞれのモジュール中のセットレジスタ（図1の231）を含むモジュール236-250のそれぞれに接続され、コプロセッサ224の全体の動作を可能とする。図2を見やすくするために、図2では制御バス231からそれぞれのモジュール236-250までの接続は示していない。

【0039】図3は、利用可能なモジュールレジスタの模式的なレイアウト260を示した図である。レイアウト260は、コプロセッサ224の全体制御のためのレジスタ261と命令制御部235とが含まれる。コプロ

セッサモジュール236-260には、同様のレジスタ262が含まれる。

3. 2 ホスト／コプロセッサ・キューイング

上述のアーキテクチャによれば、ホストプロセッサ202と画像コプロセッサ204との間での協調が十分にとられていることが必要であることがわかる。しかしながら、これに対する解は一般的なものであり、上述のアーキテクチャ特有のものではないため、以下ではより一般的な計算ハードウェア環境を想定して説明する。

【0040】現代のコンピュータシステムは、動的メモリ割当を行うために何かしらのメモリ管理手法を必要とする。1つあるいは複数のコプロセッサを有するシステムでは、コプロセッサによる動的メモリ割当とメモリ使用との間で同期をとるための手法が必要である。一般的なコンピュータハードウェア構成では、CPUと特別のコプロセッサとを備え、それぞれが一連のメモリ群を共有している。このようなシステムでは、CPUのみがメモリを動的に割り当てることのできるシステム中唯一の部位である。コプロセッサが使用するようにCPUがメモリを割り当てた時点で、コプロセッサは当該メモリが不必要になりCPUによって解放されるまで、自由にメモリを利用することができる。すなわち、コプロセッサがメモリの使用を終えた後にメモリが解放されることを保証するために、CPUとコプロセッサとの間には何かしらの同期が必要となる。この同期に関しては、種々の解決策が示されているが、必ずしも性能の面で好ましいとは言い難い。

【0041】静的に割り当てられたメモリを用いれば、同期の問題を避けることができるが、メモリ資源の利用を動的に適応させることが不可能となる。同様に、コプロセッサが処理の実行を終えるまでCPUをブロックし待たせておくことも可能であるが、並列性を失い、全体のシステム性能を犠牲にすることになる。コプロセッサからの処理の終了を知らせるインタラプト信号の利用も可能であるが、コプロセッサのスループットが非常に高い場合には大きな処理のオーバーヘッドとなってしまう。

【0042】高性能要件の他に、このようなシステムでは動的なメモリ欠乏に対してしなやかに対処しなければならない。多くのコンピュータシステムでは種々のメモリサイズ構成が可能となっているが、多くのメモリを具備するシステムでは有効資源を最大限に利用して性能を最大にすることが重要である。同様に、最小のメモリサイズ構成のシステムでは、少ないメモリながらも十分な動作を可能にすべきであり、少なくともメモリ欠乏の際には性能がしなやかに劣化すべきである。

【0043】これらの問題を解決するために、システム性能を最大にするとともに、コプロセッサのメモリ使用をシステム容量や実行する処理の複雑さに動的に適応化する同期機構が必要である。図4に、（ホスト）CPUとコプロセッサとの同期をとる好適な構成を示す。図中

の参照番号は、図1の説明において利用したものを用いている。

【0044】図4において、CPU202はシステム中のすべてのメモリ管理を統括している。CPU202が、自身、あるいはコプロセッサ224での利用のために、メモリ203を割り当てる。コプロセッサ224はグラフィックス特有の命令セットを有しており、ホストプロセッサ202と共有しているメモリ203から命令1022を実行することができる。これらの命令のそれぞれは結果1024を共有メモリ203に書き込むことができ、またメモリ203からオペランドを読み込むこともできる。ここでコプロセッサ命令のオペランド1023や結果1024を記憶するに要するメモリ203の量は、処理の複雑さや種別に依存する。

【0045】CPU202は、コプロセッサ224によって実行される命令1022を生成する処理をも行う。CPU202とコプロセッサ224との間の並列性を最大にするために、CPU202によって生成された命令は1022に示されるようにキューイングされてからコプロセッサ224において実行される。キュー1022中の各命令は、コプロセッサ224のためにホストCPU202によって割り当てられた共有メモリ203中のオペランド1023や結果1024を参照することができる。

【0046】図5に示すように、これらの処理を行うために、命令生成部1030、メモリ管理部1031、キュー管理部1032が接続されている。これらすべてのモジュールはホストCPU202上で単一プロセスとして実行される。コプロセッサ224における実行命令は命令生成部1030において生成され、メモリ管理部1031のサービスを利用して生成された命令のオペランド1023や結果1024のための領域を割り当てる。また、命令生成部1030は、キュー管理部1032のサービスを利用して、コプロセッサ224で実行する命令をキューイングする。

【0047】各命令がコプロセッサ224において実行されると、CPU202はメモリ管理部1031によって命令のオペランド用に割り当てられていたメモリを解放することができる。ある命令の結果が次の命令のオペランドとなることも可能であり、その後でCPUによってメモリが解放される。コプロセッサ224が命令を終えると同時にインタラプト信号を送出しメモリを解放するのではなく、コプロセッサ224が命令を終えた後のある時点でクリーンアップ機構を起動し、命令の処理に要した資源をシステムが解放する。クリーンアップ機構が起動される時点は、メモリ管理部1031とキュー管理部1032との関係に依存しており、利用可能なシステムメモリ量や各コプロセッサ命令に必要なメモリ量に応じて動的に適応させることができる。

【0048】図6は、コプロセッサ命令キュー1022

の構成を模式的に示した図である。命令群はホストCPU202によりペンディング命令キュー1040に挿入され、コプロセッサ224によって読み出され実行に移される。コプロセッサ224における実行処理が終了すると、命令はクリーンアップキュー1041に転送され、コプロセッサ224が処理を終えた後で命令が必要とした資源の解放を行う。

【0049】命令キュー1022自身は固定あるいは動的に変化するサイズの巡回バッファとして構成される。命令キュー1022は、CPU202による命令の生成とコプロセッサ224における命令の実行とを分離している。各命令のオペランドと結果メモリは、命令生成時に命令生成部1030からの要求に応じてメモリ管理部1031(図5)によって割り当てられる。新しく生成された命令のためのメモリ割当てが、以下で説明するメモリ管理部1031とキュー管理部1032との協調動作を起動させ、利用可能なメモリ量や命令の複雑さにシステムが自動的に適応できるようにしている。

【0050】命令キュー管理部102は、コプロセッサ224が命令生成部1030によって生成された命令を実行し終えるまで、待機することができる。しかし、メモリ管理部1031によって割り当てられる命令キュー1022とメモリ203が十分大きければ、コプロセッサ224を全く待つ必要はないか、あるいは少なくともすべての命令シーケンスが終了するまで待機する必要はない。大きなジョブではこれらの待機時間が、数分間にも及ぶため、効果は大きい。しかし、ピーク時のメモリ使用量は利用可能なメモリ量を容易に超えることもある。この時点で、キュー管理部1032とメモリ管理部1031との間で協調的な動作が開始される。

【0051】命令キュー管理部1032にとって、終了した命令を「クリーンアップ」し、動的に割り当てられたメモリを解放するようにとの指示がなされる時点は適宜で構わない。メモリ管理部1031が利用可能なメモリが少なくなりつつある、あるいはなくなったことを検出した場合には、キュー管理部1032にクリーンアップ処理を指示し、コプロセッサ224によってもはや利用されていないメモリを解放させる手段をとる。これにより、メモリ管理部1031は、CPU202がコプロセッサ224を待つ、あるいはコプロセッサ224と同期することなく、命令生成部1030からの新しく生成された命令に要するメモリ要求を満足させることができる。

【0052】メモリ管理部1031からキュー管理部1032に終了命令をクリーンアップする要求を出しても、命令生成部の新しい要求を満たすに足る十分メモリが解放されなかった場合には、メモリ管理部1031はキュー管理部1032にペンディング命令キュー1040中の処理中命令の一部、例えば半分が終了するまで待機せよ、と要求する。これにより、コプロセッサ224

命令のいくつかが終了するまでCPU202処理はブロックされることになる。コプロセッサ224命令のいくつかが終了すると、これらの命令のオペランドが解放され、要求を満たすに十分なメモリが得られる。処理中の命令の一部のみを待つことにより、少なくともいくつかの命令はペンディング命令キュー1040に存在しており、コプロセッサ224は常に動作していることになる。多くの場合、CPU202が待機するペンディング命令キュー1040中の一部をクリーンアップすることにより、メモリ管理部1031にとって十分なメモリが解放され、命令生成部1030の要求を満たすことができる。

【0053】コプロセッサ224がペンディング命令の例えば半分が実行終了するまで待機したとしても要求を満たすだけのメモリが解放されなかったという特殊なケースの場合には、メモリ管理部1031はすべてのペンディングコプロセッサ命令が終了するまで待機するという最後の手段をとる。システムの現在のメモリ容量を超えるような非常に大きな複雑なジョブなどを除いて、これにより命令生成部1030の要求を満たすに十分な資源が解放される。

【0054】このようなメモリ管理部1031とキュー管理部1032との協調動作により、システムに与えられたメモリ量203の中で効率的にスループットを最大にすることが可能となる。より多くのメモリがあれば同期の必要性は少なくなり、より大きなスループットを得ることができる。逆に、より少ないメモリの場合には、コプロセッサ224が乏しいメモリ203を使つての処理が終わるまで待機することが多くなり、利用可能なメモリが少なくても動作はするものの性能は劣化する。

【0055】命令生成部1030からの要求を満たす際にメモリ管理部1031が行う処理ステップを以下にまとめる。各ステップは順々に実行され、ステップ後にメモリ管理部1031が要求を満たすに十分なメモリ203が得られるかどうか調べる。十分なメモリが得られる場合には要求が満たされるため、ステップを終了する。得られなかった場合には、次のステップに進み、要求を満たすべくより過激な処理に進む。

1. 利用可能なメモリ203で要求を満たすことを試みる
2. すべての終了した命令をクリーンアップする
3. ペンディング命令の一部が終了するのを待つ
4. すべてのペンディング命令が終了するのを待つ

メモリ管理

```
ALLOCATE_MEMORY  
BEGIN
```

```
IF 要求を満たすのに十分なメモリが得られないとすると  
THEN 終了した命令すべてをクリーンアップ（一掃）する  
ENDIF  
IF 要求を満たすのに十分なメモリが未だ得られないとすると
```

なお、要求を満たすために、ペンディング命令のうちの異なる部分（例えば、1/3や2/3）を待機するか、多量のメモリを使用することがわかっている特定の命令を待機するなど、他のオプションを用いることもできる。

【0056】図7において、メモリ管理部1031とキュー管理部1032との間での協調動作に加えて、固定長命令キューバッファ1050が溢れた場合にはキュー管理部1032がコプロセッサ224と同期をとることもできる。このような状況を図7に示しており、ペンディング命令キュー1040は長さ10個の命令のキューとしている。付加される最新の命令が最も大きい数を有しているため、領域が溢れると最新の命令は位置9に格納される。次にコプロセッサ224に入力される命令は位置0において待機している。

【0057】領域が溢れた場合には、キュー管理部1032はコプロセッサ224がペンディング命令の例えば半分の処理を終えるまで待機する。この待機により、通常はキュー管理部1032によって挿入される新しい命令に必要な十分な領域が解放される。新しい命令をスケジューリングする際のキュー管理部1032の動作は以下の通りである。

1. 命令キュー1040に十分な領域が残っているかテストする
 2. 十分な領域が残っていない場合は、コプロセッサがある所定数の命令が終了するまで待機する
 3. 新しい命令をキューに挿入する
- ある命令が終了するのを待機せよと指示されたキュー管理部1032の動作は以下の通りである。

1. 命令が終了したとコプロセッサ224から指示されるまで待機する
 2. クリーンアップされていない終了した命令がある場合には、次に終了した命令をキューから削除する
- 新しい命令を生成する際の命令生成部1030の動作は以下の通りである。

1. 命令オペランド1023に必要なメモリをメモリ管理部1031に要求する
2. 転送する命令を生成する
3. コプロセッサ命令をキュー管理部1032に転送し実行する

以上の動作プロセスを擬似コードの形で示した例を以下に示す。

【0058】

THEN WAIT_FOR_INSTRUCTIONを呼び出し、ペンディング命令の半分の終了を待つ

ENDIF

IF 要求を満たすのに十分なメモリが未だ得られないとすると

THEN エラーを出力し戻る

ENDIF 割り当てたメモリを戻す

キュー管理

SCHEDULE_INSTRUCTION

BEGIN

IF 命令キューに十分な領域が得られないとすると

THEN ある所定数の命令をコプロセッサが終了するまで待機する

ENDIF 新しい命令をキューに付加する

END

WAIT_FOR_INSTRUCTION (i)

BEGIN

命令 i が終了したとコプロセッサから指示されるまで待機する

WHILE 終了しているもののクリーンアップされていない命令が

ある

DO

IF 次の終了した命令にクリーンアップ機能が備わっている

THEN クリーンアップ機能呼び出す

ENDIF キューから終了した命令を削除する

DONE

END

命令生成部

GENERATE_INSTRUCTIONS

BEGIN

ALLOCATE_MEMORYを呼び出し、命令オペランドに必要なメモリをメモリ管理部において割り当てる

転送する命令を生成する

SCHEDULE_INSTRUCTIONを呼び出し、コプロセッサ命令をキュー管理部に転送し実行する

END

3. 3 コプロセッサのレジスタの説明

図1と3において説明したように、コプロセッサ224は各命令ストリームを実行するために複数のレジスタを備える。

【0059】図2中のモジュールに対して、表1はコプロセッサ224において用いられるレジスタの名前、種

別、説明を示しており、付録Bはそれぞれのレジスタの各フィールドを説明している。

レジスタの説明

【0060】

【表1A】

表 1: レジスタ一覧

レジスタ名	タイプ	説明
外部インターフェースコントローラレジスタ		
eic_cfg	Config2	コンフィグレーション
eic_stat	Status	ステータス
eic_err_int	Interrupt	エラーおよび割り込みステータス
eic_err_int_en	Config2	エラーおよび割り込みイネーブル
eic_test	Config2	テストモード
eic_gen_pob	Config2	総称的バスのプログラマブル出力ビット
eic_high_addr	Config1	二重アドレスサイクルオフセット
eic_wtlb_v	Control2	TLB Invalidate/Write 用仮想アドレスと操作ビット
eic_wtlb_p	Config2	TLB Write 用の物理アドレスと制御ビット
eic_mmu_v	Status	変換された Most Recent MMU 仮想アドレスと現在の LRU 位置
eic_mmu_v	Status	MMU でフェッチされた Most Recent ページテーブルの物理アドレス
eic_ip_addr	Status	PCI Bus にアクセスするための Most Recent IBus 用物理アドレス
eic_rp_addr	Status	PCI Bus にアクセスするための Most Recent RBus 用物理アドレス
eic_ig_addr	Status	総称的 Bus にアクセスするための Most Recent IBus 用アドレス
eic_rg_data	Status	総称的 Bus にアクセスするための Most Recent RBus 用アドレス
レジスタ名	タイプ	説明
ローカルメモリーコントローラレジスタ		
lmi_cfg	Control2	総称的コンフィグレーションレジスタ
lmi_sts	Status	総称的ステータスレジスタ
lmi_err_int	Interrupt	エラー及びインタラプトステータスレジスタ
lmi_err_int_en	Control2	エラー及びインタラプトイネーブルレジスタ
lmi_dcfg	Control2	DRAM コンフィグレーションレジスタ
lmi_mode	Control2	SDRAM モードレジスタ

【0061】

【表1B】

レジスタ名	タイプ	説明
周辺インターフェースコントローラレジスタ		
pic_cfg	Config2	コンフィグレーション
pic_stat	Status	ステータス
pic_err_int	Interrupt	割込み/エラーステータス
pic_err_int_en	Config2	割込み/エラーイネーブル
pic_abus_cfg	Control2	ABus 用コンフィグレーションと制御
pic_abus_addr	Config1	ABus transfer 用開始アドレス
pic_cent_cfg	Control2	Centronics 用コンフィグレーションと制御
pic_cent_dir	Config2	Centronics ピンダイレクト・コントロールレジスタ
pic_reverse_cfg	Control2	逆 (入力) データ転送用コンフィグレーションと制御
pic_timer0	Config1	初期データタイマ値
pic_timer1	Config1	事象データタイマ値

レジスタ名	タイプ	説明
その他モジュールレジスタ		
mm_cfg	Config2	コンフィグレーションレジスタ
mm_stat	Status	ステータスレジスタ
mm_err_int	Interrupt	エラー及び割込みレジスタ
mm_err_int_en	Config2	エラー及び割込みマスク
mm_gefg	Config2	グローバル・コンフィグレーションレジスタ
mm_diag	Config	自己診断コンフィグレーションレジスタ
mm_grst	Config	グローバルリセットレジスタ
mm_gerr	Config2	グローバルエラーレジスタ
mm_gexp	Config2	グローバル例外レジスタ
mm_gint	Config2	グローバル割込みレジスタ
mm_active	Status	グローバルアクティブ信号

【0062】

【表1C】

レジスタ名	タイプ	説明
インストラクションコントローラレジスタ		
ic_cfg	Config2	コンフィグレーションレジスタ
ic_stat	Status/ Interrupt	ステータスレジスタ
ic_err_int	Interrupt	エラー及び割込みレジスタ(エラー解除のためと割込み用の番込み)
ic_err_int_en	Config2	エラー及び割込みイネーブルレジスタ
ic_ipa	Control1	ストリーム命令レジスタ
ic_tda	Config1	ストリーム Todo レジスタ
ic_fna	Control1	ストリーム終了レジスタ
ic_inta	Config1	ストリーム割込みレジスタ
ic_loa	Status	Last Overlapped Instruction Sequence (最後にオーバーラップした命令シーケンス) 番号のストリーム
ic_ipb	Control1	B ストリーム命令ポインタ
ic_tdb	Config1	B ストリーム Todo レジスタ
ic_fnb	Control1	B ストリーム終了レジスタ
ic_intb	Config1	B ストリーム割込みレジスタ
ic_lob	Status	Last Overlapped Instruction Sequence (最後にオーバーラップした命令シーケンス) 番号のBストリーム
ic_sema	Status	A ストリームセマフォ
ic_semb	Status	B ストリームセマフォ

レジスタ名	タイプ	説明
データキャッシュコントローラレジスタ		
dcc_cfg1	Config2	DCC コンフィグレーション1 レジスタ
dcc_stat	Status	状態器ステータスビット
dcc_err_int	Status	DCC エラーステータスレジスタ
dcc_err_int_en	Control1	DCC エラー割込みイネーブルビット
dcc_cfg2	Control2	DCC コンフィグレーション2 レジスタ
dcc_addr	Config1	特別アドレスモード用ベースアドレスレジスタ
dcc_lv0	Control1	ライン 0~31 用の"valid" ビットステータス
dcc_lv1	Control1	ライン 32~63 用の"valid" ビットステータス
dcc_lv2	Control1	ライン 64~95 用の"valid" ビットステータス
dcc_lv3	Control1	ライン 96~127 用の"valid" ビットステータス
dcc_raddrb	Status	オペランドオーガイナイザ B 要求アドレス
dcc_raddrc	Status	オペランドオーガイナイザ C 要求アドレス

【0063】

【表1D】

レジスタ名	タイプ	説明
dcc_test	Control1	DCC テストレジスタ
ピクセルオーガナイザレジスタ		
po_cfg	Config2	コンフィグレーションレジスタ
po_stat	Status	ステータスレジスタ
po_err_int	Interrupt	エラー/割込みステータスレジスタ
po_err_int_en	Config2	エラー/割込みイネーブルレジスタ
po_dmr	Config2	データ操作レジスタ
po_subst	Config2	入替え値レジスタ
po_cdp	Status	現データポインタ
po_len	Control1	長さレジスタ
po_said	Control1	開始アドレス或は即時データ
po_idr	Control2	画像サイズレジスタ
po_muv_valid	Control2	MUV 有効ビット
po_muv	Config1	MUV RAM のベースアドレス

【0064】

【表1E】

レジスタ名	タイプ	説明
オペランドオーガナイザBレジスタ		
oob_cfg	Config2	コンフィグレーションレジスタ
oob_stat	Status	ステータスレジスタ
oob_err_int	Interrupt	エラー/割込みレジスタ
oob_err_int_en	Config2	エラー/割込みイネーブルレジスタ
oob_dmr	Config2	データ操作レジスタ
oob_subst	Config2	入替え値レジスタ
oob_cdp	Status	現データレジスタ
oob_len	Control1	入力長レジスタ
oob_said	Control1	オペランド開始アドレス
oob_tile	Control1	タイリング長/オフセットレジスタ
オペランドオーガナイザCレジスタ		
ooc_cfg	Config2	コンフィグレーションレジスタ
ooc_stat	Status	ステータスレジスタ
ooc_err_int	Interrupt	エラー/割込みレジスタ
ooc_err_int_en	Config2	エラー/割込みイネーブルレジスタ
ooc_dmr	Config2	データ操作レジスタ
ooc_subst	Config2	入替え値レジスタ
ooc_cdp	Status	現データポインタ
ooc_len	Control1	入力長レジスタ
ooc_said	Control1	オペランド開始アドレス
ooc_tile	Control1	タイリング長/オフセットレジスタ

レジスタ名	タイプ	説明
JPEG 符号化器レジスタ		
jc_cfg	Config2	コンフィグレーション
jc_stat	Status	ステータス
jc_err_int	Interrupt	エラー及び割込みステータスレジスタ
jc_err_int_en	Config2	エラー及び割込みイネーブルレジスタ
jc_rsi	Config1	リスタート間隔
jc_decode	Control2	現命令の復号
jc_res	Control1	残りの値
jc_table_sel	Control2	復号した命令によるテーブル選択

【0065】

【表1F】

メインデータバスレジスタ		
mdp_cfg	Config2	コンフィグレーション
mdp_stat	Status	ステータス
mdp_err_int	Interrupt	エラー/割込み
mdp_err_int_en	Config2	エラー/割込みイネーブル
mdp_test	Config2	テストモード
mdp_op1	Control2	現操作 1
mdp_op2	Control2	現操作 2
mdp_por	Control1	正のオペレータ用オフセット
mdp_bi	Control1	索引テーブルの開始/オフセットのブレンド
mdp_bm	Control1	ブレンド終了或は行列の行及び列の数, 2 進数の位値, 及びハーフトーンの階調数
mdp_len	Control1	ブレンド長

レジスタ名	タイプ	説明
リザルトオーガナイザレジスタ		
ro_cfg	Config2	コンフィグレーションレジスタ
ro_stat	Status	ステータスレジスタ
ro_err_int	Interrupt	エラー/割込みレジスタ
ro_err_int_en	Config2	エラー/割込みイネーブルレジスタ
ro_dmr	Config2	データ操作レジスタ
ro_subst	Config1	入替え値レジスタ
ro_cdp	Status	現データレジスタ
ro_len	Status	出力長レジスタ
ro_sa	Config1	開始アドレス
ro_idr	Config1	画像サイズレジスタ
ro_vbase	Config1	コプロセッサ仮想ベースアドレス
ro_cut	Config1	出力カットレジスタ
ro_lmt	Config1	出力長レジスタ
PCIバスコンフィグレーションスペースエイリアス		
		A read only copy of PCI コンフィグレーション空間レジスタ 0x0 を 0xD と 0xF に読出し専用コピー
pci_external_cfg	Status	リセット時、外部シリアル ROM からダウンロードした 32 ビットフィールド。コプロセッサの動作に影響しない。

【0066】

【表 1 G】

レジスタ名	タイプ	説明
入力インターフェーススイッチレジスタ		
iis_cfg	Config2	コンフィグレーションレジスタ
iis_stat	Status	ステータスレジスタ
iis_err_int	Interrupt	割込み/エラーステータスレジスタ
iis_err_int_en	Config2	割込み/エラーイネーブルレジスタ
iis_ic_addr	Status	ICからの入力アドレス
iis_doc_addr	Status	DCCからの入力アドレス
iis_po_addr	Status	POからの入力アドレス
iis_burst	Status	PO, DCC & ICからのバースト長
iis_base_addr	Config1	ホストメモリマップにおけるコプロセッサメモリオブジェクトのベースアドレス
iis_test	Config1	テストモードレジスタ

【0067】これらのレジスタ中で着目すべきものは以下のものである。

(a) 命令ポインタレジスタ (ic_ipa と ic_ipb)。これらのレジスタペアは現在実行している命令

の仮想アドレスを格納する。仮想アドレスの昇順に命令がフェッチされ実行される。制御が不連続な仮想アドレスに移る場合にはジャンプ命令が用いられる。各命令には、32ビットのシーケンス番号が付与され、シーケンス番号は一命令ごとに1ずつ増える。シーケンス番号はコプロセッサ224とホストCPU202双方において、命令の生成と実行の同期をとるために用いられる。

(b) 終了レジスタ (ic_fnaとic_fnb)。これらのレジスタペアは、終了した命令のシーケンス番号を格納する。

(c) ToDoレジスタ (ic_tdaとic_tdb)。これらのレジスタペアは、キューイングされている命令のシーケンス番号を格納する。

(d) インタラプトレジスタ (ic_intaとic_intb)。これらのレジスタペアは、インタラプトをかけるシーケンス番号を格納する。

(e) インタラプト状態レジスタ (ic_stat.a_primeとic_stat.b_prime)。これらのレジスタペアは、インタラプト、終了レジスタとが合致した時点でインタラプトを起動するフラグであるブライムビットを格納する。本ビットは、インタラプト状態 (ic_stat) レジスタ中の他のインタラプトイネーブルビットや他の状態／構成情報と同様に格納される。

(f) レジスタアクセスセマフォア (ic_semaとic_semb)。ホストCPU202は、コプロセッサ224への高速性、即ち、1回以上のレジスタへの書き込みを必要とするレジスタアクセスに先立ちセマフォアを入手しておかなければならない。これに対して、高速性を必要としないレジスタアクセスの場合は何時でも実行することができる。ホストCPU202がセマフォアを入手することに付随する欠点は、現在実行中の命令が終了するまでコプロセッサの実行が中断することである。レジスタアクセスセマフォアは、コプロセッサ224の構成／状態レジスタの1ビットとして構成される。これらのレジスタは命令制御部のレジスタ領域中に存在する。前述の通り、コプロセッサの各サブモジュールは、それぞれ構成／状態レジスタを備えており、通常の命令実行においてレジスタが設定される。これらのすべてのレジスタは、レジスタマップ上に表されており、多くは命令実行において暗黙的に修正される。ホストはレジスタマップを介してこれらのレジスタの内容を知ることができる。

3.4 複数ストリームフォーマット

前述の通り、資源を最大限に有効に利用するために、また外部周辺装置に高速に出力するために、コプロセッサ224は2つの独立な命令ストリームの1つを実行する。通常は、1つの命令ストリームは出力デバイスが適時点が必要とする現在の出力ページに対応しており、2つ目の命令ストリームが他の命令ストリームが休止中で

あるときにコプロセッサ224のモジュールを利用する。ここで、最も重要な点は、必要な出力データを適時点で出力することであるとともに、続くページ、バンドなどの準備のために資源を最大限に利用することである。従って、コプロセッサ224は、全く独立であるものの同じように実行される2つの命令ストリーム（以下、AとBと呼ぶ）を実行するように設計される。命令はホストCPU202上で動作しているソフトウェアによって生成され、ラスト画像アクセラレータカード220に転送されコプロセッサ224によって実行されることが望ましい。通常動作では、命令ストリームの1つ（ストリームA）は、他の命令ストリーム（ストリームB）よりも高い優先度で動作する。命令ストリームあるいはキューはホストRAM203（図1）中の一つあるいは複数のバッファに書き込まれる。バッファは開始時点で割り当てられ、アプリケーションの実行中はホスト203の物理メモリに固定される。各命令はホストRAM203の仮想メモリ環境に格納されることが好ましく、ラスト画像コプロセッサ224が仮想アドレスから物理アドレスへの変換を行い、次の命令の位置としてホストRAM203中の対応する物理アドレスを決定する。これらの命令は順々にコプロセッサ224のローカルメモリに格納される。

【0068】図8は、ホストRAM203中に格納されている2つのストリームAとBのフォーマットを示す図である。ストリームAとBそれぞれのフォーマットは本質的に同一である。コプロセッサ224における簡単な実行モデルは、以下のものから構成される。

*AストリームとBストリームの2つの命令仮想ストリーム

*通常はある時点で1つのみの命令が実行される

*どちらかのストリームが優先権を有することもできるし、「ラウンドロビン」的に優先権を交互にすることもできる

*どちらかのストリームを「ロック」して、ストリーム優先権や他のストリームの命令実行可能度に関わらず、確実に実行することもできる

*どちらかのストリームが空であっても良い

*どちらかのストリームが利用不能であっても良い

*どちらかのストリームは、後続の命令が「オーバラップ」していなければ、次の命令の実行と「オーバラップ」しているような命令を含んでも良い

*各命令は32ビットの1つずつ増加するような「一意な」シーケンス番号を有する

*各命令はインタラプトや命令実行を停止させるコードを有していても良い

*外部インタフェースの遅延の影響を最小限にするために、命令をあらかじめフェッチしても良い

命令制御部235は、コプロセッサ224の全体の実行制御を行うためや、必要な時にホストRAM203から

命令をフェッチするために、コプロセッサの命令実行モデルを実装している。一つの命令ごとに、命令制御部235は命令の復号を行い、CBus231を介してモジュール中の種々のレジスタを構成し、該当モジュールに命令を実行させる処理を行う。

【0069】図9は、命令制御部235で実行する命令実行サイクルを簡単な形で示した図である。命令実行サイクルは4つの主なステージ276-279から成る。第1ステージ276では、命令ストリームにおいて命令がペンディング状態であるかどうかを調べる。ペンディング状態である場合には、命令をフェッチして277、復号ならびに実行し278、レジスタを更新する279。

3. 5 現在のアクティブストリームの決定

第1ステージでは、2つのステップを実行しなければならない。

1. 命令がペンディングしているかどうかの決定
 2. どの命令ストリームを次にフェッチするか
- どの命令がペンディングであるかを決定するためには次

```
if エラーモードでなく、稼働モードであり、バイパスモードでもなく、自己診断モードである
```

```
if Aストリームがロックされていて休止中でない
```

```
if Aストリームが稼働モードであり、かつ「Aストリームのシーケンス番号が休止中、あるいはAストリームに命令が存在する」
```

```
命令はペンディングしている
```

```
else 命令はペンディングしていない
```

```
end if
```

```
else if Bストリームがロックされていて休止中でない
```

```
if Bストリームが稼働モードであり、かつ「Bストリームのシーケンス番号が休止中、あるいはBストリームに命令が存在する」
```

```
命令はペンディングしている
```

```
else 命令はペンディングしていない
```

```
end if
```

```
else /*ストリームがロックされていない*/
```

```
if Aストリームが稼働モードで休止中でない、かつ「Aストリームのシーケンス番号が休止中、あるいはAストリームに命令が存在する」
```

```
命令はペンディングしている
```

```
else 命令はペンディングしていない
```

```
end if
```

```
end if
```

```
else /*インタフェース制御部が稼働していない*/
```

```
命令はペンディングされていない
```

```
end if
```

いかなる命令もペンディングしていない場合には、命令制御部235はペンディング命令が見つかるまで「スピン」あるいはアイドル状態となる。

【0071】どのストリームがアクティブであるか、どのストリームを次に実行するかを決定するために、次の状態が調べられる。

1. どちらかのストリームがロックされているか
2. AとBのストリームにどの優先権が付与されてお

の可能性を調べる。

1. 命令制御部がイネーブルかどうか
2. 内部エラーやインタラプトにより命令制御部が休止しているかどうか
3. ペンディングしている外部エラー状態があるかどうか
4. AあるいはBのストリームがロックしているかどうか
5. どちらかのストリームシーケンス番号がイネーブルかどうか
6. どちらかのストリームがペンディング命令を有しているかどうか

以下に示す擬似コードは、上記ルールに基づいて命令がペンディングしているかどうかを決定するアルゴリズムを示したものである。このアルゴリズムは、既知の技術を用いて、命令制御部235中に状態遷移機械を介してハードウェアとして実装することができる。

【0070】

り、最後に実行した命令ストリームはどちらであるか

3. どちらかのストリームが稼働しているか
4. どちらかのストリームがペンディング命令を有しているか

以下は、命令制御部によって実装される擬似コードを示

したものであり、どのように次にアクティブとなるストリームを決定するかを示している。 【0072】

```
if Aストリームがロックされている
次のストリームはA
else if Bストリームがロックされている
次のストリームはB
else /*どちらのストリームもロックされていない*/
if Aストリームが稼動モード、かつ「Aストリームのシーケンス番号
が休止中、あるいはAストリームに命令が存在する」、かつ「Bストリームが稼
動モードで、「Bストリームのシーケンス番号が休止中、あるいはBストリーム
に命令が存在」」しなければ、次のストリームはA
else if Bストリームが稼動モード、かつ「Bストリームのシーケ
ンス番号が休止中、あるいはBストリームにペンディング命令が存在する」、か
つ「Aストリームが稼動モードで、「Aストリームのシーケンス番号が休止中、
あるいはAストリームに命令が存在」」しなければ、次のストリームはB
else /*どちらのストリームも命令が存在しない*/
if pri=0 /*A高、B低*/
次のストリームはA
else if pri=1 /*A低、B高*/
次のストリームはB
else if pri=2 or 3 /*ラウンドロビン*/
if 最後のストリームがA
次のストリームはB
else
次のストリームはA
end if
end if
end if
end if
```

条件は常に変化しているため、すべての条件を短時間で調べることが必要である。

3. 6 現在のアクティブストリームのフェッチ命令
次のアクティブ命令ストリームを決定すると、命令制御部235は対応する命令ポインタレジスタ（ic_ipaとic_ipb）中のアドレスを用いて命令をフェッチする。しかしながら、有効な命令が既に命令制御部235中のプレフェッチバッファ内に存在する場合には、命令制御部235は命令をフェッチしない。

【0073】以下の条件が満たされるときに、プレフェッチバッファ中の命令が有効になる。

1. プレフェッチバッファが有効である
 2. プレフェッチバッファ中の命令が現在のアクティブストリームと同じストリームからのものである
- プレフェッチバッファの内容の有効性は、ic_statレジスタ中のプレフェッチビットによって表され、当該ビットは命令のプレフェッチが成功した際にセットされる。なお、命令制御部235のいかなるレジスタへの外部書き込みも、プレフェッチバッファの内容を無効にさせる。

3. 7 復号、実行命令

命令がフェッチされ、受理されると、命令制御部235は命令を復号し、命令を実行するためにコプロセッサ224のレジスタ229を構成する。

【0074】ラスト画像コプロセッサ224において用いられる命令フォーマットは、命令の生成がホストCPU202からの命令によって実行され、ホストに対して直接的なオーバーヘッドになるという点で、従来のプロセッサ命令セットとは異なる。また、命令はホストRAM203に格納され、図1のPCIバス206を介してコプロセッサ224に転送されるため、命令はできるだけ小型化すべきである。好ましくは、コプロセッサ224は単一の命令によって実行開始されることが望ましい。また、将来の変更に最大限対処可能とするためには、命令セットの柔軟性をできるだけ保持することが望ましい。更に、コプロセッサ224において実行される命令はオペランドデータの長いストリームにも適用でき、最適な性能が得られるようにすることも好ましい。なお、コプロセッサ224が用いる命令復号「哲学」として、「一般的な命令」の復号を簡潔にかつ高速に行うととも

に、「一般的でない」処理に対してもコプロセッサ224の動作に対して細かい制御をホストシステムが行えるようにデザインを取り入れている。

【0075】図10は、それぞれが32ビットの8ワードから成る単一命令280フォーマットを示している。各命令は、命令ワード（オペコード）281、オペランドの種別を示すオペランドあるいは結果タイプデータワード282を含む。3つのオペランドA、B、Cのアドレス283-285も、結果アドレス286とともに含まれる。更に、領域287も、ホストCPU202が用いる命令に関する情報を格納するために含んでいる。

【0076】図11は、命令の命令オペコード281の構造290を示した図である。命令オペコードは32ビット長で、主オペコード291、補オペコード292、インタラプト（I）ビット293、一部復号（Pd）ビット294、レジスタ長（R）ビット295、ロック（L）ビット296、長さ297を含む。命令ワード290のそれぞれのフィールドの説明を以下の表に示す。

【0077】オペコード説明

【0078】

【表2A】

表2：オペコード

フィールド	説明
major opcode [3..0] (主オペコード)	命令のカテゴリ 0: Reserved 1: 総称的色空間の変換 2: JPEG 圧縮と伸長 3: 行列演算 4: 画像コンボリューション 5: 画像変換 6: データ符号化 7: ハーフトーン 8: 階層画像復号 9: メモリコピー 10: 内部レジスタとメモリアクセス 11: 命令流れ制御 12: 合成 13: 合成 14: Reserved 15: Reserved
minor opcode [7..0] (補オペコード)	命令の詳細。このフィールドの符号化は major opcode による。
I	1 = 完了時に割込みと停止 0 = 完了時に割込みと停止なし

【0079】

【表2B】

フィールド	説明
pd	部分的復号 1 = "partial decode" 機構を使用する 0 = "partial decode" 機構を使用しない
R	1 = Pixel Organizer's 入力長レジスタ(po_len)により特定された命令長 0 = オペコード長フィールドにより特定された命令長
L	1 = この命令ストリーム(A or B)は次の命令のために "locked"される 0 = この命令ストリーム(A or B)は次の命令のために "locked"されない
length [15..0] (長さ)	読出し或は発生されるデータ数

【0080】I ビットフィールド293をセットすることによって、命令が終了した時点で命令の実行がインタラプトされ休止するように命令をコード化することができる。なお、このインタラプトは「命令終了インタラプ

ト」と呼ばれる。一部復号ビット294は、一部復号ビット294のビットがセットされ、ic_cfgレジスタ中で稼働モードになると、以下に述べるように命令の実行に先立ち種々のモジュールがマイクロコード化され

るというような一部復号機能を提供する。ロックビット 296 は、開始にあたり 1 つ以上の命令を必要とする処理の際に用いられる。この際には、命令に先立ち種々のレジスタがセットされ、次の命令のために現在の命令ストリームを「ロック」される。L ビット 296 がセットされると、命令が終了した時点で次の命令が同じストリームからフェッチされる。長さフィールド 297 は各命令の一般的な定義であり、必要となる「入力データ項目」数あるいは「出力データ項目」数として定義され、16 ビット長である。64、000 項目以上の入力データ項目のストリームに対する処理の場合には、R ビット

295 がセットされ、図 2 のピクセルオーガナイザ 246 中の `po_len` レジスタから入力長を得る。当該レジスタはこのような命令の直前にセットされる。

【0081】図 10 において、ある命令に必要なオペランド 283～286 の数は用いる命令タイプに応じて可変である。以下の表は、各命令タイプごとにオペランド数と長さの定義とを示したものである。

オペランドタイプ

【0082】

【表 3】

表 3: オペランドのタイプ

インストラクションクラス	長さを決定するもの		オペランドの数
合成	input	pixels	3
総称的色空間の変換	input	pixels	2
JPEG 圧縮と伸長	input	bytes	2
他の圧縮と伸長	input	bytes	2
画像変換とコンボリューション	output	bytes	2
行列演算	input	pixels	2
ハーフトーン	input	pixels,bytes	2
メモリコピー	input	pixels,bytes	1
階層画像復号	input	pixels,bytes	1 又は 2
流れ制御	fixed	fixed	2
内部アクセス命令	fixed	fixed	4

【0083】図 12 は、3 オペランド命令に対する図 10 のデータワード、オペランド記述子 282 のデータワードフォーマット 300 と、2 オペランド命令に対するデータワードフォーマット 301 とを示している。以下

の表に、オペランド記述子のコード化の詳細を示す。

オペランド記述子

【0084】

【表 4】

表 4:オペランド

フィールド	説明
what	0=命令特別モード 記述子の残りフィールドは主オブコードとともに解釈されることを示す。サポートされている命令特別モードは、 主オブコード=0-11:予約 マイナーオブコード=12-13:(合成):オペランドCがビットマップ減衰率であることを意味する。Occ_dmrレジスタはcc=1, normalize=0として適切にセットされる。 主オブコード=14-15:予約 1:逐次的アドレッシング 2:タイルアドレッシング 3:一定データ
L	0=短い:直接データ 1=長い:データへのポインタ
if	内部フォーマット 0=ピクセル 1=アンパックバイト 2=パックバイト 3=他
S	0=当該オペランドに適するようにデータ操作レジスタを設定 1=データ操作レジスタをそのまま使用
C	0=キャッシュ不可能 1=キャッシュ可能 注意:一般に、オペランドをキャッシュ可能とすることで性能の向上を図ることができる。(逐次データなどのように)局所的に参照される確率が低いようなオペランドであっても、データをパースティックにホストコンピュータに効率的に転送することができるため、キャッシュによる利点が生じる。
P	外部フォーマット 0=アンパックバイト 1=バックストリーム
bo[2:0]	ビットオフセット。ビットごとのデータ開始バイト中のオフセットを指定
R	0=オペランドCは設定するレジスタを示さない 1=オペランドCは設定するレジスタを示す オペランドが3つ以下である命令のときのみに関連するビットである

【0085】上述の表において、一定データアドレスモードの場合には、コプロセッサ224が1つの内部データ項目をフェッチあるいは計算して、この項目を当該オペランドの命令長として用いる。タイルアドレスモードの場合には、コプロセッサ224がいくつかのデータをサイクルして「タイル効果」を得る。オペランド記述子のLビットがゼロの場合には、データが短く、データ項目がオペランドワード中に存在することを意味する。

【0086】図10において、それぞれのオペランド/結果ワード283-286は、オペランド自身の値あるいはデータが格納されているオペランド/結果の開始位置を示す32ビット仮想アドレスを含む。図2の命令制御部235は、命令を二段階で復号する。最初に、命令の主オブコードが有効であるかを調べ、主オブコード

(図11)が無効である場合にはエラーを生成する。次に、CBus231を介して種々のレジスタを設定することにより、命令制御部235が命令を実行し、命令に

指定されている動作を行う。なお、設定するレジスタがないような命令もある。

【0087】各モジュールのレジスタは動作に応じていくつかの種別に分けられる。まず、状態レジスタタイプがあり、他のモジュールからは「読み込まれるのみ」で、レジスタを含むモジュールによって「読み込み/書き込み」されるものがある。次に、構成レジスタの一番目のタイプ(以降、config1)は、モジュールから外部的に「読み込み/書き込み」され、レジスタを含むモジュールからは「読み込みのみ」される。これらのレジスタは一般にアドレス値などの大きなタイプ構成情報を格納する際に用いられる。構成レジスタの二番目のタイプ(以降、config2)はすべてのモジュールから読み込み、書き込みができるが、レジスタを含むモジュールからは読み込みしかできない。このレジスタタイプは、レジスタのビットごとのアドレッシングが必要なときに用いられる。

【0088】制御タイプのレジスタとしては種々のものが存在する。第一のタイプ（以降、control1レジスタ）はすべてのモジュール（レジスタを含むモジュールも含む）によって読み込み／書き込みが可能である。Control1レジスタは、アドレス値などの大きな制御情報を格納する際に用いられる。同様に、制御レジスタの第二のタイプ（以降、control2）は、ビットごとに設定される。

【0089】最後のレジスタタイプ（インタラプトレジスタ）は、レジスタを含むモジュールによって1にセットされ、セットされたビットに「1」を外部から書き込みことによりゼロにリセットすることができるようなビットをレジスタ内に含む。このようなタイプのレジスタはそれぞれのモジュールからのインタラプト／エラー信号に対処するために用いられる。

【0090】コプロセッサ224の各モジュールは、命令を実行中でビジー状態のときには、CBus231上のc_activeラインをセットする。このため、命令制御部235は、CBus231上の各モジュールからのc_activeラインの「OR」をとり、命令が終了した時点を把握することができる。ローカルメモリ制御モジュール236と周辺インタフェース制御モジュール237とは、オーバラップ命令を実行することができ、オーバラップ命令を実行する際に起動するc_backgroundラインを備える。オーバラップ命令は、ローカルメモリインタフェースと周辺インタフェースとの間でデータを転送する「ローカルDMA」命令である。

【0091】オーバラップローカルDMA命令の実行サイクルは、他の命令の実行サイクルとは異なる。オーバラップ命令が実行に移されるにあたっては、命令制御部235が既にオーバラップ命令が実行されているかどうかを調べる。オーバラップ命令が既に存在すれば、あるいはオーバラップ命令が不稼動モードになっていれば、命令制御部235は命令が終了するのを待ってから、当該命令の実行に移る。オーバラップ命令が存在せず、かつ稼動モードになっていれば、命令制御部235はすぐにオーバラップ命令を復号し、周辺インタフェース制御部237やローカルメモリ制御部236を構成し命令を実行する。レジスタを構成し終えたら、従来の意味で命令が終了するのを待たずに命令制御部235はレジスタ（終了レジスタ、状態レジスタ、命令ポインタ等）を更新する。この時点で、終了シーケンス番号はインタラプトシーケンス番号と同一であれば、「オーバラップ命令終了」、インタラプト信号を出力するのではなく単に当該信号を用意する。「オーバラップ命令終了」インタラプト信号は、オーバラップ命令が完全に終了した時点で出力される。

【0092】命令が復号されると、命令制御部は現在の命令を実行しつつ、次の命令をプレフェッチする。ほと

んどの命令では、命令のフェッチ、復号よりも命令の実行に要する時間の方がかなり長い。命令制御部235は、以下の条件が揃った時点で命令をプレフェッチする。

1. 現在実行中の命令がインタラプトや休止中でない
2. 現在実行中の命令がジャンプ命令でない
3. 次の命令ストリームがプリフェッチ可能である
4. 他にペンディングしている命令が存在する

命令制御部235がプレフェッチ可能と判断すると、次の命令に要求を出し、プレフェッチバッファに配置し、バッファを有効にする。ここまで処理を進めると、命令制御部235は現在実行中の命令が終了するまでは何もすることがなく、当該命令の終了をCBus231上のc_activeとc_backgroundラインを調べることをのみを行う。

3. 8 命令制御部のレジスタの更新

命令が終了すると、命令制御部235は新しい状態を反映させるためにレジスタの更新を行う。この処理は外部からのアクセスとの同期の問題を避けるために高速に行わなければならない。この高速更新処理は以下の手順で行われる。

1. 適切なレジスタアクセスセマフォアの入手。セマフォアが命令制御部235の外部のエージェントによって占有されている場合には、セマフォアが解放されるまで命令実行サイクルが待機し、解放されてから処理に移る。
2. 適切なレジスタの更新。命令が適切なジャンプ命令でない場合には、命令ポインタ（ic_ipaとic_ipb）を命令のサイズ分増加させる。ジャンプ命令のときは、ジャンプ先の値が命令ポインタにロードされる。従って、シーケンス番号が稼動モードであれば終了レジスタ（ic_fnaとic_fnb）は増加することになる。

【0093】状態レジスタ（ic_stat）も新しい状態を反映させるように適切に更新される。必要であれば、休止ビットを設定することもある。インタラプトが生じ、インタラプトに対する休止が稼動状態になったり、エラーが生じた場合には、命令制御部235は休止する。休止は、状態レジスタ中の命令ストリーム休止ビット（a_pauseとb_pause）をセットすることによって起動される。命令実行を再開する際には、これらのビットを0にリセットしなければならない。

3. 1クロックサイクル時間、CBus231上にc_end信号を送出し、コプロセッサ224中の他のモジュールに命令が終了した旨を伝える。

4. 必要であればインタラプトを送出する。インタラプトの送出手は、以下の状況のときに送出手される。

- a. 「シーケンス番号終了」インタラプトが生じたとき。すなわち、終了レジスタ（ic_fnaとic_fnb）シーケンス番号がインタラプトシーケンス番号と

一致したとき。このとき、インタラプトが準備され、シーケンス番号が稼動モードになり、インタラプトが生じる。あるいは、

b. 終了した命令が終了時点でインタラプトするように符号化されている場合。この場合にはインタラプト機構が起動される。

3. 9 レジスタアクセスセマフォアのセマンティックス

レジスタアクセスセマフォアは、複数の命令制御レジスタに高速アクセスを提供する機構である。高速アクセスを必要とするレジスタとして、以下のものが挙げられる。

1. 命令ポインタレジスタ (ic__ipaとic__ipb)
2. ToDoレジスタ (ic__tdaとic__tdb)
3. 終了レジスタ (ic__fnaとic__fnb)
4. インタラプトレジスタ (ic__intaとic__intb)
5. 構成レジスタ中の休止ビット (ic__cfg)

外部エージェントはすべてのレジスタをいつでも安全に読むことができる。また、外部エージェントはすべてのレジスタにいつでも書き込むことができるが、命令制御部235がこれらのレジスタ中の値を更新してしまわないように、外部エージェントはまずレジスタアクセスセマフォアを入手しなければならない。命令制御部は、レジスタアクセスセマフォアが外部で宣言されている間は上述のレジスタ中の値を更新することはできない。また、命令制御部235は、高速を維持するために1クロックサイクルの間に上述のすべてのレジスタを更新する。

【0094】前述のように、シーケンス機構が稼動モードであれば、各命令には32ビットの「シーケンス番号」が付与されている。命令シーケンス番号は順々に増加していき、0xFFFFFFFFから0x00000000にラッピングされる。外部からの書き込みがインタラプトレジスタ (ic__intaとic__intb) になされると、命令制御部235はすぐに以下の比較と更新を行う。

1. インタラプトシーケンス番号 (インタラプトレジスタ中の値) が同一ストリームの終了シーケンス番号 (終了レジスタ中の値) よりも「大きければ」(モジュロ演算)、命令制御部は状態レジスタ中の「シーケンス番号終了」準備ビット (ic__stat中のa__primedとb__primedビット) をセットすることで「シーケンス番号終了」インタラプト機構を準備する。
2. インタラプトシーケンス番号が終了シーケンス番号よりも「小さく」、当該ストリームにおいてオーバーラップ命令が実行中であり、インタラプトシーケンス番号が最後のオーバーラップ命令シーケンス番号 (ic__loaあるいはic__lobレジスタ中の値) と同一であれ

ば、命令制御部はic__statレジスタ中のa__ol__primedあるいはb__ol__primedビットをセットすることで「オーバーラップ命令シーケンス番号終了」インタラプト機構を準備する。

3. インタラプトシーケンス番号が終了シーケンス番号よりも「小さく」、当該ストリームにおいてオーバーラップ命令が実行中であり、インタラプトシーケンス番号が最後のオーバーラップ命令シーケンス番号と同一でなければ、インタラプトシーケンス番号は終了命令を示すことになり、インタラプト機構は準備されない。

4. インタラプトシーケンス番号が終了シーケンス番号よりも「小さく」、当該ストリームにおいてオーバーラップ命令が実行中でなければ、インタラプトシーケンス番号は終了命令を示すことになり、インタラプト機構は準備されない。

【0095】外部のエージェントは、状態レジスタ中のインタラプト準備ビット (a__primed, a__ol__primed, b__primed, b__ol__primedビット) をセットすることができ、インタラプト機構を独立に起動、解除することができる。

3. 10 命令制御部

図13は、命令制御部235をより詳細に示した図である。命令制御部235は、命令実行サイクルを処理しコプロセッサ224の全体の実行制御を管理する実行制御部305を含む。実行制御部305は、命令制御部235の全体の実行制御を管理し、命令シーケンスを決定し、命令のフェッチやプレフェッチを行い、命令の復号や命令制御レジスタの更新を行う。命令制御部は更に命令復号器306を備える。命令復号器306は、プレフェッチバッファ307から命令を受信し、前述の通り復号する。命令復号器306は、他のコプロセッサモジュール中のレジスタを構成して命令を実行する処理も行う。プレフェッチバッファ制御部307は、プレフェッチバッファ制御部中のプレフェッチバッファからの読み込みや書き込みを管理するとともに、命令復号器306と入力インタフェーススイッチ252 (図2) との間のインタフェースをも管理する。また、プレフェッチバッファ制御部307は二つの命令ポインタレジスタ (ic__ipaとic__ipb) の更新をも管理する。命令制御部235、種々のモジュール239 (図2)、外部インタフェース制御部238 (図2) からのCBus231 (図2) へのアクセスは、三つのモジュールのアクセス要求間での調停を行う「CBus」調停部308において行われる。要求はCBus231によって種々のモジュールのレジスタ部に転送される。

【0096】図14は、図13の実行制御部305をより詳細に示した図である。前述の通り、実行制御部は図9の命令実行サイクル275の処理を管理し、特に以下の処理を行う。

1. 次の命令をどの命令ストリームから取り出すかを決

定し、

2. 当該命令のフェッチを開始し、
3. プレフェッチバッファに格納されている命令の復号を命令復号器に指示し、
4. 次の命令のプレフェッチを決定して開始し、
5. 命令の終了を決定し、
6. 命令が終了したらレジスタを更新する。

【0097】実行制御部は、全体の命令実行サイクルを管理する大きなコア状態器310（以下、中枢部と呼ぶ）を備える。図15は、上述の命令実行サイクルを管理する中枢部310状態遷移図を示した図である。図14において、実行制御部は命令プレフェッチ論理部311を備える。この部位は、実行すべき命令が存在するかどうか、どの命令ストリームに命令が属するか、の決定処理を行う。図15の遷移図において開始312ならびにプレフェッチ313状態は、この情報を用いて命令を入手する。図14のレジスタ管理部317は、双方の命令ストリームのレジスタアクセスセマフォアをモニタし、各モジュール中の必要なすべてのレジスタを更新する処理を行う。また、終了レジスタ（ic_fnaとic_fnb）とインタラプトレジスタ（ic_intaとic_intb）とを比較し、「シーケンス番号終了」インタラプトを行うべきかどうかを決定する処理も、レジスタ管理部317が行う。更に、レジスタ管理部317はインタラプト準備処理も行う。オーバーラップ命令部318は、ic_statレジスタ中の適切な状態ビットの管理を通して、オーバーラップ命令の終了処理の管理を行う。実行制御部は、更に中枢部310と図13の命令復号器306との間のインタフェースを行う復号インタフェース部319を備える。

【0098】図16は、命令復号部306をより詳細に示した図である。命令復号器はコプロセッサを構成してプレフェッチバッファ内の命令を実行する処理を行う。命令復号器306は、多くの小さな状態マシンの組み合わせである大きな状態マシンから構成される命令復号シーケンサ321を備える。命令シーケンサ321は、各モジュール中のレジスタをセットするCBUSディスパッチャ312と通信する。また、命令復号シーケンサ321は、命令の有効性や命令のオーバーラップ状況などの

関連情報を実行制御部に伝える。ここで、命令の有効性チェックは命令オPCODEが予約されているオPCODEであるかどうかをチェックするものである。

【0099】図17は、図16の命令ディスパッチャシーケンサ321をより詳細に示した図である。命令ディスパッチャシーケンサ321は、全体のシーケンス制御状態マシン324と連続したモジュール毎構成シーケンサ状態マシン（例えば325や326）を備える。モジュール毎構成シーケンサ状態マシンは構成すべき各モジュールに与えられる。全体として状態マシンはモジュールのコプロセッサマイクロプログラミングを定義する。状態マシン（例えば325）は、CBUSディスパッチャに全体のCBUSを利用して種々のレジスタをセットするように指示し、処理のための種々モジュールを構成する。特定のレジスタに書き込みをするためには、命令の実行が開始されなければならない。一般に命令の実行にはシーケンサ321が処理のためにコプロセッサのレジスタを構成する以上の時間が必要である。付録Aにおいて、コプロセッサの命令シーケンサによって実行されるマイクロプログラミング処理と命令シーケンサ321によってセットアップされた形式を示す。

【0100】実際には、命令復号シーケンサ321は命令ごとにコプロセッサ中のすべてのモジュールを構成するわけではない。以下の表では、命令クラスに対するモジュール構成順序を、ピクセルオーガナイザ246（PO）、データキャッシュ制御部240（DCC）、オペランドオーガナイザB247（OOB）、オペランドオーガナイザC248（OOC）、主データパス242（MDP）、結果オーガナイザ249（RO）、JPEGエンコーダ241（JC）などの構成されるモジュールとともに示している。なお、外部インタフェース制御部238（EIC）、ローカルメモリ制御部236（LMC）、命令制御部235自身（IC）、入力インタフェーススイッチ252（IIS）、雑多モジュール（MM）などのモジュールは、命令復号処理中には構成されることはない。

【0101】モジュール立ち上げ順序

【0102】

【表5】

表 5: Module Setup Order

インストラクションクラス	モジュールコンフィグレーション シーケンス	シーケンス ID
合成	PO, DCC, OOB, OOC, MDP, RO	1
CSC	PO, DCC, OOB, OOC, MDP, RO	2
JPEG 符号化	PO, DCC, OOB, OOC, JC, RO	3
データ符号化	PO, DCC, OOB, OOC, JC, RO	3
変換とコンボリユーション	PO, DCC, OOB, OOC, MDP, RO	2
行列演算	PO, DCC, OOB, OOC, MDP, RO	2
ハーフトーン	PO, DCC, OOB, MDP, RO	4
総称的メモリーコピー	PO, JC, RO	8
周辺 DMA	PIC	5
階層画像-水平補間	PO, DCC, OOB, OOC, MDP, RO	6
階層画像-その他	PO, DCC, OOB, OOC, MDP, RO	4
内部アクセス	RO, RO, RO, RO	7
その他	-	-

【0103】図17において、各モジュール構成シーケンサ（例えば325）は必要なレジスタアクセス処理を行って特定のモジュールを構成するように管理する。また、全体のシーケンス制御状態マシン324は、前述の順序でモジュール構成シーケンサの全体の動作を管理する。図18は、上の表に従って関連するモジュール構成シーケンサを起動する全体シーケンス制御を状態遷移図330で表した図である。各モジュール構成シーケンサは、モジュールの実行中に種々のレジスタをセットするために、CBus ディスパッチャを制御して、レジスタ内容を変更する処理を行う。

【0104】図19は、図13のプリフェッチバッファ制御部307をより詳細に示した図である。プリフェッチバッファ制御部は単一のコプロセッサ命令（6×32ビットワード）を格納するためのプリフェッチバッファ335を備える。そして、プリフェッチバッファはIBusシーケンサ336によって制御される一つの書き込みポートと、命令復号器、実行制御部、命令制御部CBusインタフェースにデータを送出する一つの読み込みポートを備える。IBusシーケンサ336は、プリフェッチバッファ335の入力インタフェーススイッチへの接続においてバスプロトコルを監視する。また、命令をフェッチするためにアドレスを生成するアドレス管理部337をも備える。アドレス管理部337は、ic_ipaあるいはic_ipbの一つを選択し入力インタフェーススイッチへのバスに接続する機能と、最後の命令がどのストリームからフェッチされたかに基づいてic_ipaあるいはic_ipbの一つを増加させる機能と、ic_ipaとic_ipbレジスタにジャンプ

先のアドレスを格納する機能とを有する。PBC制御部339はプリフェッチバッファ制御部307の全体の制御を行う。

3. 11 モジュールローカルレジスタファイルの説明
図13に示したように、命令制御モジュール自身を含む各モジュールは、図20に示してあるCBusインタフェース制御部303とともに上述したレジスタ304の内部セットを備え、CBus要求を受け付けるとともに当該要求に応じて内部レジスタを更新する処理を行う。モジュールの制御は、CBusインタフェース302を介してモジュール中のレジスタ304に書き込むことによって行われる。CBus調整部308（図13）は、命令制御部235、外部インタフェース制御部、雑多モジュールのどのモジュールがCBusを制御し、CBusのマスターとして動作し、レジスタの書き込み/読み出しを行うのかを決定する。

【0105】図20は、各モジュールにおいて用いられるCBusインタフェース303の標準構成を示した図である。標準CBusインタフェース303はCBus302からの読み出し要求や書き込み要求を受信するとともに、モジュール内の種々のサブモジュールによって341を介して更新されるレジスタファイル304を備える。更に、メモリ領域の読み出しを含むサブモジュールのメモリ領域の更新を行う制御ライン344が備わっている。標準CBusインタフェース303はCBusの目的地として振る舞い、レジスタ304や他のサブモジュールのメモリオブジェクトの読み出し要求や書き込み要求を受け付ける。

【0106】「c_reset」信号345は標準CB

usインタフェース103内のすべてのレジスタをデフォルト状態にセットする。しかし、「c_reset」は自身とCBusマスターとの間の信号のやり取りを制御する状態マシンはリセットしない。そのため、「c_reset」がCBus処理中に送出されたとしても、当該処理は何かしらの形で終了することになる。「c_

int」347、「c_exp」348、「c_err」349信号は、以下の式に基づいてモジュールerr_intとerr_int_enレジスタの内容より生成される。

【0107】

【数1】

$$c_err = \sum_{\text{error}[i] \text{ not reserved}} \text{error}[i] \text{ AND } err_mask[i]$$

【0108】

【数2】

$$c_int = \sum_{\text{interrupt}[i] \text{ not reserved}} \text{interrupt}[i] \text{ AND } int_mask[i]$$

【0109】

【数3】

$$c_exp = \sum_{[i] \text{ not reserved}} \text{exception}[i] \text{ AND } exp_mask[i]$$

【0110】信号「c_sdata_in」と「c_svalid_in」345は、モジュール列の中での前のモジュールからのデータ／有効信号であり、信号「c_sdata_out」と「c_svalid_out」350は、モジュール列の中での次のモジュールへのデータ／有効信号である。標準CBusインタフェース303の機能としては以下のものが含まれる。

1. レジスタの読み出し／書き込み管理
2. メモリ領域の読み出し／書き込み管理
3. テストモードの読み出し／書き込み管理
4. サブモジュールの監視／更新管理
3. 12 レジスタ読み出し／書き込み管理

標準CBusインタフェース303はCBus上に流れるレジスタ読み出し／書き込み要求やビットセット要求を受け付ける。標準CBusインタフェースが管理するCBus命令として以下の2種類ある。

1. タイプA

タイプAは、他のモジュールが標準CBusインタフェース303内のレジスタに1、2、3、4バイト読み出し／書き込みする動作をする。書き込み動作では、命令サイクルの直後のクロックサイクルでデータサイクルが生じる。なお、レジスタ書き込み／読み出しのタイプフィールドはそれぞれ「1000」と「1001」である。標準CBusインタフェース303は命令を復号して、命令がモジュールのアドレスを指しているか、読み出し／書き込み動作のどちらかであるか、を調べる。読み出し動作では、標準CBusインタフェース303は、CBus処理の「reg」フィールドを用いてどのレジスタ出力に「c_sdata」バス350を接続するかを選択する。書き込み動作では、標準CBusインタフェース303は「reg」フィールドと「byte」フィールドを用いて選択されたレジスタにデータを書き込む。読み出し動作が終了すると、標準CBusイ

ンタフェースはデータを戻すと同時に「c_svalid」350を送出する。書き込み動作が終了すると、標準CBusインタフェース303は「c_svalid」350を送出して返答する。

2. タイプC

タイプCは、1つのレジスタ中のバイトの1つに他のモジュールが1ビットあるいは複数ビット書き込む動作をする。命令とデータとは1つのワードにまとめられる。

【0111】標準CBusインタフェース303は命令をチェックして、命令がモジュールのアドレスを指しているかを調べる。また、「reg」「byte」「enable」フィールドを復号して、必要なイネーブル信号を生成する。また、命令のデータフィールドを取り出し、取り出したデータをワードの4バイトすべてに転送する。これにより、必要なビットはすべてのイネーブルバイト中のイネーブルビットに書き込まれることになる。この動作においては返答は必要ない。

3. 13 メモリ領域読み出し／書き込み管理

標準CBusインタフェース303はCBus上のメモリ読み出し／書き込み要求を受け付ける。メモリ読み出し／書き込み要求を受け付けると、標準CBusインタフェース303は要求がモジュールのアドレスを指しているかを調べる。そして、命令のアドレスフィールドを復号することで、標準CBusインタフェースは適切なアドレスと、メモリ読み出し／書き込みを行うサブモジュールへのアドレスストロブ信号344とを生成する。書き込み動作では、標準CBusインタフェースは、命令からのバイトイネーブル信号をサブモジュールに転送する。

【0112】標準CBusインタフェース303の動作は、CBus302上のCBus命令のタイプフィールドを復号し、次のサイクルにおいてデータがレジスタファイル304に取り込まれるか、あるいは他のサブモジ

ジュール344に転送されるようにするために、レジスタファイル304と出力セクタ353に適切なイネーブル信号を生成するような読み出し/書き込み制御部352によって制御される。CBus命令がレジスタ読み出し動作であれば、読み出し/書き込み制御部352は出力セクタ353をイネーブルにし、「c_sdataバス」345への正しいレジスタ出力を選択する。命令がレジスタ書き込み動作であれば、読み出し/書き込み制御部352はレジスタファイル304をイネーブルにし、次にサイクルでデータを選択する。もしその命令がメモリエリアのリード/ライトであれば、読み出し/書き込み制御部352は適切な信号344を生成し、モジュールが管理するメモリ領域を制御する。レジスタファイル304は、レジスタ選択復号部355、出力セクタ353、インタラプト356、エラー357、例外358生成部、アンマスクエラー生成部359、あるモジュールのレジスタを構成するレジスタ部360の4つの部位から構成される。レジスタ選択復号部355は、読み出し/書き込み制御部352からの信号「ref_en」(レジスタファイルイネーブル)「write」「reg」を復号し、あるレジスタをイネーブルにするためのレジスタイネーブル信号を生成する。出力セクタ353は、読み出し/書き込み制御部352からの信号「reg」出力に応じて、レジスタ読み出し処理のために正しいレジスタデータを選択しc_sdate_outラインに出力する。

【0113】例外生成部356～359は入力中にエラーが検出されたら出力エラー信号(例えば、347～349、362)を生成する。各出力エラーを計算する手法は前述の通りである。レジスタ部360は、表5においてレジスタセットの構成を説明したときに論じたように、要求に応じて種々のタイプになり得る。

3.14 CBus構成

前述の通り、CBus(制御バス)は、各モジュールの標準CBusインタフェース中のレジスタをセットするための情報を転送することによって、全体的に各モジュールを制御する。標準CBusインタフェースの記述から明らかなように、CBusは以下の二つの目的を有する。

1. 各モジュールを駆動する制御バス
2. RAM, FIFO, 各モジュール中の状態情報のためのアクセスバス

CBusは命令-アドレス-データプロトコルを用いて、モジュール中の構成レジスタをセットすることにより、モジュールを制御する。一般に、レジスタは各命令ごとにセットされるが、修正はどの時点でも行うことができる。CBusは状態情報や他の情報を集め、データを要求することにより種々のモジュールからRAMやFIFOデータにアクセスする。

【0114】CBusは以下の3つのどちらかにより処理ごとに駆動される。

1. 命令実行時の命令制御部235(図2)
2. ターゲット(スレーブ)モードバス動作実行時の外部インタフェース制御部238(図2)
3. 外部CBusインタフェースが構成された際には外部デバイス

いずれの場合でも、駆動モジュールはCBusの発モジュールとなり、他のすべてのモジュールが可能な着モジュールとなる。バスの調整処理は命令制御部が行う。

【0115】以下の表は、好適な実施例において用いるのに適しているCBus信号の一つの定義を示したものである。

CBus信号定義

【0116】

【表6】

表 6: CBus 信号定義

Name	Type	Definition
c_iad[31:0]	source	instruction-address-data
c_valid	source	CBus instruction valid
c_sdata[31:0]	destination	status/read data
c_svalid	destination	status/read data valid
c_reset[15:0]	source	reset lines to each module
c_active[15:0]	destination	active lines from each module
c_background[15:0]	destination	background active lines from each module
c_int[15:0]	destination	interrupt lines from each module
c_error[15:0]	destination	error lines from each module
c_rep1, c_req2	ETC, external	bus control request
c_gnt1, c_gnt2	IC	bus control grant
c_end	IC	end of instruction
clk	global	clock

【0117】 CBus の c_iad 信号はアドレスデータを含み、二つの異なるサイクルにおいて制御部によって駆動される。

1. c_iad 上で CBus 命令やアドレスが駆動される命令サイクル (c_valid 高)
2. c_iad (書き込み動作) や c_sdata (読み出し動作) 上でデータが駆動されるデータサイクル (c_valid 低)

書き込み動作の場合は、命令に関するデータは命令サイクルの直後に c_iad バス上に置かれる。読み出し動作の場合は、データサイクルが終了するまで読み出し動作のターゲットモジュールが c_sdata 信号を駆動する。

【0118】 図 21 において、バスは 32 ビットの命令・アドレス・データフィールドを含む。このフィールドは以下の 3 つのタイプ (370~372) がある。

1. タイプ A 動作 (370) は、コプロセッサ中のレジスタや各モジュールのデータ領域の読み出し/書き込みを行うために用いられる。これらの動作は、ターゲットモード PCI サイクルを実行している外部インタフェース制御部 238、命令のためにコプロセッサを構成している命令制御部 231、外部 CBus インタフェースに

よって生成される。

【0119】 これらの動作では、命令サイクルの直後のクロックサイクルがデータサイクルとなる。

2. タイプ B 動作 (371) は診断モードで用いられ、ローカルメモリにアクセスしたり、一般インタフェース上のサイクルを生成する。これらの動作は、ターゲットモード PCI サイクルを実行している外部インタフェース制御部や外部 CBus インタフェースによって生成される。データサイクルは命令サイクルの後のどの時点でも良く、データサイクルは c_svalid 信号を用いて着モジュールから返答される。

3. タイプ C 動作 (372) はモジュールのレジスタ中の各ビットをセットするために用いられる。これらの動作は、命令のためにコプロセッサを構成している命令制御部 231 や外部 CBus インタフェースによって生成される。タイプ C 動作ではデータサイクルはなく、データは命令サイクル中に含まれる。

【0120】 各命令のタイプフィールドは、以下の表に従って関連する CBus 処理を符号化したものである。

CBus 処理タイプ

【0121】

【表 7】

表 7: CBus トランザクションタイプ

c_iad.type の値	トランザクションタイプ	フォーマットタイプ
0000	no-op	A, B, C
0001	reserved	
0010	peripheral interface write(周辺インターフェース書き込み)	B
0011	peripheral interface read(周辺インターフェース読取り)	B
0100	generic bus write(総称的バス書き込み)	B
0101	generic bus read(総称的バス読取り)	B
0110	local memory write(ローカルメモリ書き込み)	B
0111	local memory read(ローカルメモリ読取り)	B
1000	register write(レジスタ書き込み)	A
1001	register read(レジスタ読取り)	A
1010	module memory write(モジュールメモリ書き込み)	A
1011	module memory read(モジュールメモリ読取り)	A
1100	test mode write(テストモード書き込み)	A
1101	test mode read(テストモード読取り)	A
1110	bit set (ビットセット)	C
1111	reserved	

【0122】バイトフィールドは、レジスタ中のビットをセットするために用いられる。モジュールフィールドはCBus上の命令のアドレス先モジュールを指定するフィールドである。レジスタフィールドはモジュール中のどのレジスタを更新するかを指定するフィールドである。アドレスフィールドは、動作を行うメモリ部位を指定するフィールドである、RAM、FIFOなどのアドレスを指定するものである。イネーブルフィールドは、ビット設定命令が用いられたときに選択されたバイト中の選択されたビットをイネーブルにするフィールドである。データフィールドは、更新されるべきバイトに書き込まれるビットデータを含む。

【0123】前述の通り、CBusは各モジュールごとに、モジュールが命令実行中のときに送出されるc_activeラインを含む。命令制御部はこの信号に基づいて命令の終了時を知ることができる。また、CBusは各モジュールごとにバックグラウンドモード時に動作するc_backgroundラインを、リセット、エラー検出、インタラプトを行うためのリセット、エラー、インタラプトラインとともに含む。

3.15 コプロセッサデータタイプとデータ操作
図2において、コプロセッサ部224の動作、特にJPEG符号化器241や主データバスのコプロセッサ中の主な計算処理動作を簡潔にするため、コプロセッサは外部フォーマットと内部フォーマットとを差別化するデータモデルを用いる。外部データフォーマットは、ローカルメモリインタフェースやPCIバスなどのコプロセッサの外部インタフェースに現われるデータフォーマット

である。逆に、内部データフォーマットは、コプロセッサ224の主機能モジュール間で現われるフォーマットである。図22は、種々の入力/出力フォーマットを模式的に示した図である。入力外部フォーマット381は、ピクセルオーガナイザ246、オペランドオーガナイザB247、オペランドオーガナイザC248への入力フォーマットである。これらのオーガナイザは、入力外部フォーマットを、JPEG符号化器241や主データバス部242へ入力される入力内部フォーマット382に再フォーマットする。また、これら2つの機能部は出力データを出力内部フォーマットで出力し、結果オーガナイザ249が出力内部フォーマットを所望出力フォーマット304に変換する。

【0124】実施例では、外部データフォーマットは3つのタイプに分けられる。第一のタイプは、データごとに4つまでのチャンネルを有し、各チャンネルが1、2、4、8、あるいは16ビットサンプルから成り立っているような連続ストリームから成るデータの「バックストリーム」である。バックストリームは、ピクセル、ピクセルに変換されるデータ、まとめられたビットなどを表現する際に用いられる。また、コプロセッサはリトルエンディアンバイトアドレッシングとバイト中ではビッグエンディアンビットアドレッシングを用いる。図23はバックストリームフォーマットの第一の例を示している。ここでは、各オブジェクト387は、各チャンネルごとに2ビットのチャンネル0、チャンネル1、チャンネル2の三つのチャンネルから構成される。このフォーマットのデータ配置が388である。図24の次の例390では、

各データオブジェクトが32ビットワードを有し、チャンネルごとに8ビット有する4チャンネルオブジェクト395が示されている。図25の第三の例395では、ビットアドレス397から始まるチャンネルごとに8ビットを有するチャンネルオブジェクト396が示されている。もちろん、アプリケーションに応じて、データチャンネルの実際の幅や数は変化する。

【0125】外部データフォーマットの第二のタイプは「アンパックバイトストリーム」であり、各ワード中の1バイトのみが有効であるような32ビットワードのシーケンスである。このフォーマットの例が図26の399として示されており、各ワード中の単一バイト400のみが用いられる。さらなる外部データフォーマットは「他」フォーマットとして分類されるオブジェクトで表現される。一般に、これらのデータオブジェクトは色空間変換表、ハフマン符号化表などの大きな表型のデータである。

【0126】コプロセッサは4つの内部データタイプを用いる。第一のタイプは「パックバイト」フォーマットであり、最後の32ビットワードを除いて4アクティブバイトの32ビットワードから成るフォーマットである。図27に、ワードが4バイトであるパックバイトフォーマットの例402を示す。図28に示す次のデータタイプは「ピクセル」フォーマットであり、4アクティブバイトチャンネルの32ビットワード403から成るフォーマットである。このピクセルフォーマットは4つのチャンネルデータとして解釈される。

【0127】図29に示す次の内部データタイプは「アンパックバイト」フォーマットであり、各ワードは一つのアクティブバイトチャンネル405と三つの非アクティブバイトチャンネルから成るフォーマットである。この際、アクティブバイトチャンネルは最小バイトを占める。他の内部データオブジェクトは「他」データフォーマットとして区分される。外部フォーマットの入力データは適切な内部フォーマットに変換される。図30は、種々のオーガナイザによって実行される外部フォーマット410から入力フォーマット411への変換形態を示している。図31は、結果オーガナイザ249によって実行される内部フォーマット412から外部フォーマット413への変換形態を示している。

【0128】以下、変換を実行する処理をより詳細に説明する。まず入力データ外部フォーマットから内部フォーマットへの変換であるが、図32は変換処理において種々のオーガナイザによって用いられる手法を示している。はじめは外部他フォーマット416であるが、これは種々のオーガナイザを経ずに単に通過する。次に、外部アンパックバイトフォーマット417は、アンパック正規化418を行って内部アンパックバイトと呼ばれるフォーマット419を生成する。アンパック正規化418処理は、外部アンパックバイトストリームから非アク

ティブ3バイトを取り除く処理を行う。図33はアンパック正規化処理を示したものであるが、4バイトチャンネルを有する入力のうち1つのバイトチャンネルのみが出力フォーマット419において有効な結果となっており、単なるバイトを出力している様子を示している。

【0129】図32において、パック正規化421処理は、外部パックストリーム422中の要素オブジェクトをバイトストリーム423に変換する処理を行う。チャンネルの各要素のサイズがバイト以下であれば、サンプルは8ビット値に補間される。例えば、4ビット単位をバイト単位に変換する場合には、4ビット値0xNはバイト値0xNNに変換される。1バイト以上のオブジェクトの場合には切り捨てが行われる。ストリーム422でサポートされる入力オブジェクトサイズは、1、2、4、8、16ビットサイズである。なお、これらは、本発明が適用されるシステム中のデータオブジェクトやワードの全幅に依存する。

【0130】図34は、チャンネルごとに（図23のデータフォーマット386ごとくのように）2ビット有する3チャンネルオブジェクト形式の入力データ422が入力されたときのパック正規化421の様子を示している。出力データはバイトチャンネルフォーマット423になっている。この際、必要であれば各チャンネルに「補間処理」が施され、8ビットサンプルが生成される。

【0131】図32において、ピクセルストリームはその後、パック処理425、アンパック処理426、要素選択処理427のいずれかに送られる。図35はパック処理425の例を示したもので、単に非アクティブバイトチャンネルが取り除かれ、ワードごとの4アクティブバイトにパックされたバイトストリームが生成される様子を示している。即ち、単一の有効バイトストリーム430がワードごとの4アクティブバイトを有するフォーマット431に圧縮される。アンパック処理426はほぼパック処理の反対の処理であり、アンパックバイトがワードの最小バイトとなる。図36は、パックバイトストリーム433がアンパックされ結果434が得られる様子を示している。

【0132】図37は要素選択427処理を示したものであり、Nを単位ごとの入力チャンネル数とすると、入力ストリームからN要素を選択する処理である。アンパック処理は「プロトタイプピクセル」、例えば437を生成するときに用いられる。なお、ピクセルチャンネルは最小バイトから埋められる。図38は、形式436の入力データが要素選択部427によって変換され、プロトタイプピクセルフォーマット437が生成される様子を示している。

【0133】要素選択が行われると、要素入替処理440（図32）が行われる。図38は要素入替処理の様子を示したもので、内部データレジスタ441に格納された一定値で選択要素を入れ替え、例のように出力要素2

4 2を生成する様子を示している。図3 2において、処理段4 2 5、5 2 6、4 4 0の出力はレンスワップ処理4 4 4に送られる。図3 9に示されているように、レンスワップ処理はあるレーンを他のレーンにバイトごとに多重化する処理であり、あるレーンを他のレーンに複製する処理をも含む。図3 8の例では、チャンネル3とチャンネル1とを入れ替え、チャンネル3をチャンネル2とチャンネル1に複製する様子が示されている。

【0 1 3 4】図3 2において、レンスワップ処理4 4 4が終わると、データストリームが再読み出しされて複製処理4 4 6に移る前に、マルチユースト値RAM 2 5 0に格納されることもある。複製処理4 4 6は単にデータオブジェクトを複製する処理である。図4 0は、複製処理4 4 6をピクセルデータに適用した様子であり、複製ファクターは1である。

【0 1 3 5】図4 1は、複製処理をパックバイトデータに適用した様子である。図4 2は、出力内部フォーマット3 8 3から出力外部フォーマット3 8 4にデータを変換する結果オーガナイザ2 4 9の処理を示したものである。この処理では、図3 2に示した変換処理と同様の処理4 2 4、4 2 5、4 2 6、4 4 0を含むが、処理4 5 0では更に要素非選択4 5 1、非正規化4 5 2、バイトアドレッシング4 5 3、書き込みマスキング4 5 4の処理を含んでいる。図4 3に示した要素非選択処理4 5 1は、図3 7の要素選択処理の逆処理であり、不必要なデータが削除される。例えば、図4 3では、入力中の3つの有効チャンネルのみが取り出され、データ項目4 5 6にパックされる。

【0 1 3 6】図4 4に示した非正規化処理は、図3 4で示したパック正規化処理4 2 1のほぼ反対の動作をする。非正規化処理では、バイト単位で扱われていた各オブジェクトあるいはデータ項目を非バイト値に変換する処理が行われる。図4 2のバイトアドレッシング処理4 5 3は、バイトアドレッシングに必要なバイトごとの再構成処理を行う。外部アンパックバイト出力ストリームでは、ストリームアドレスの最小2ビットがアクティブストリームに対応する。バイトアドレッシング処理4 5 3では、外部アンパックバイトが用いられているとき(図4 5)、1つのバイトチャンネルから他のチャンネルバイトに

出力ストリームが再マップされる。外部パックストリームが用いられているときは(図4 6)、バイトアドレッシングモジュール4 5 3は出力ストリームの開始アドレスを図示のように再マップする。

【0 1 3 7】図4 2の書き込みマスク処理4 5 4を図4 7に示す。書き込みされないパックストリームのあるチャンネル(例えば4 6 0)をマスクする処理である。適用される入力/出力データタイプ変換は、以下のデータ操作レジスタの内容に基づいて決められる。

*ピクセルオーガナイザデータ操作レジスタ(p o _d m r)

*オペランドオーガナイザBとオペランドオーガナイザCデータ操作レジスタ(o o r _d m r, o o c _d m r)

*結果オーガナイザデータ操作レジスタ(r o _d m r)

命令のための各データ操作レジスタの設定は、以下の2つの方法によってなされる。

1. 命令実行の直前にコプロセッサレジスタに書き込む標準手法を用いて設定される
2. 現在の命令に基づいてコプロセッサ自身で設定される

命令復号処理では、コプロセッサはデータの命令ワードやデータワードの内容を調べ、種々のデータ操作レジスタをどのように設定するかを決定する処理を他の処理とともに行う。なお、命令とオペランドのすべての組み合わせが有効であるわけではない。いくつかの命令ではオペランドフォーマットを規定しているものもある。不適切なオペランドを含む命令の場合、「定義されていない」結果が生成されることになるが、エラーを生じることなく終了してしまうこともある。対応するデータ記述子の「S」ビットが0であれば、コプロセッサはデータ操作レジスタをセットし、現命令を反映させる。

【0 1 3 8】図4 8はデータ操作レジスタのフォーマットを示した図である。以下の表は、図4 8に示されたレジスタ中の種々のビットフォーマットを示している。

データ操作レジスタフォーマット

【0 1 3 9】

【表8 A】

表8: データ操作レジスタフォーマット

フィールド	説明
ls3	バイト3(最大バイト)のレインスワップ
ls2	バイト2のレインスワップ
ls1	バイト1のレインスワップ
ls0	バイト0のレインスワップ
suben	入替イネーブル 1=このバイトにおいて内部データレジスタからのデータを入れ替える 0=このバイトにおいて内部データレジスタからのデータを入れ替えない
replicate	複製カウント 生成する付加データ数を示す
wrmask	書き込みマスク 0=対応するバイトチャネルへの書き込み 1=対応するバイトチャネルへの書き込みなし
cmsb	最大/最小ビットの選択 0=非正規化処理を実行する際にバイトの最小バイトを選択(ハーフトーン処理に有用) 1=非正規化処理を実行する際にバイトの最大ビットを選択(入力正規化の逆処理に有用)
normalize	正規化ファクター: バイトに移動するビット数を示す 0=1ビットデータオブジェクト 1=2ビットデータオブジェクト 2=4ビットデータオブジェクト 3=8ビットデータオブジェクト 4=16ビットデータオブジェクト
bo	ビットオフセット: バイトよりも小さいオブジェクトの開始ビットアドレスを示す。ビットアドレスはビッグエンディアン形式である。
P	外部フォーマット 0=アンパックバイト 1=バックストリーム
if	内部フォーマット 0=ピクセル 1=アンパックバイト 2=バックバイト 3=他

【0140】

【表8B】

フィールド	説明
cc	チャネルカウント 入力オーガナイザでは、要素選択において各内部データワードを構成するための正規化入力バイト数を示す。 出力オーガナイザでは、出力データを構成するための内部データワードからの有効バイト数を示す 0=4アクティブチャネル 1=1アクティブチャネル 2=2アクティブチャネル 3=3アクティブチャネル
L	中間データ 0=短い: 中間データ 1=長い: データへのポインタ
what	アドレスモード 0=命令特定モード 1=シーケンシャルアドレッシング 2=タイルアドレッシング 3=定データ、内部データが作成され、繰り返し使用される

【0141】 各1つの命令において、複数の内部/外部 データタイプが用いられることがある。オペランド、結

果、命令タイプのすべて組み合わせは有効ではあるが、これらの組み合わせの一部のみが意味のある結果を生成する。各命令に対して期待されるオペランドと結果データタイプの具体的な組み合わせを表9に示す。表9は、外部／内部フォーマットにおいて期待されるデータタイ

プをまとめたものである。

【0142】期待されるデータタイプ

【0143】

【表9】

表 9: Expected Data Types

Instruction	Operand A (Pixel Organizer)		Operand B (Operand Organizer B)		Operand C (Operand Organizer C)		Result (Result Organizer)	
合成	ps	px	ps	px(T) bl(B)	ps ub const	ub	px ub	ps ub
GCSC	ps ift	ift	mcsc scsc (B)	mcsc scsc (B)	mcsc scsc (B)	mcsc scsc (B)		
JPEG 圧縮	ps us	pb	et (B)	et (B)	et (B)	et (B)	ub	ps
JPEG 伸長	ps	pb	fdt sdt (B)	fdt sdt (B)	fdt sdt (B)	fdt sdt (B)	pb	ps ub
Data coding	ps ub	px pb ub	et fdt sdt (B)	et fdt sdt (B)	et fdt sdt (B)	et fdt sdt (B)	px pb ub	ps ub
変換・畳み込み	skd fkcd	skd lkcd	it (B)	it (B)	it (B)	it (B)	px	ps ub
行列乗算	ps ub	px	mm (B)	mm (B)	mm (B)	mm (B)	px	ps ub
ハーフトーン	ps ub	px pb ub	ps ub	px pb ub	-	-	px pb ub	ps ub
階層画像水平補 間	ps ub	px pb ub	-	-	-	-	px pb ub	ps ub
階層画像垂直補 間	ps ub	px pb ub	ps ub	px pb ub	-	-	px pb ub	ps ub
総称的画像コピ ー	ps ub	px pb ub	-	-	-	-	px pb ub	ps ub
周辺DMA	-	-	-	-	-	-	-	-
内部アクセス	-	-	-	-	-	-	-	-
フロー制御	-	-	-	-	-	-	-	-

【0144】なお、表9において用いたシンボルは以下の通りである。

シンボルの説明

【0145】

【表10】

表 10: シンボル

シンボル	説明
ps	packed stream (パックされたストリーム)
pb	packed bytes (パックされたバイト)
ub	unpacked bytes (パックされていないバイト)
px	pixels (画素)
bl	blend (混合)
const	constant (定数)
msc	4 output channel (4出力チャネル)
cscc	1 output channel color conversion table (1出力カラー変換テーブル)
ift	Interval and Fraction tables (間隔及び部分テーブル)
ct	JPEG encoding table (JPEG 符号化テーブル)
fdt	fast JPEG decoding table (高速 JPEG 復号テーブル)
sdt	slow JPEG decoding table (低速 JPEG 復号テーブル)
skd	short kernel descriptor (短いカーネル記述子)
lkd	long kernel descriptor (長いカーネル記述子)
mm	matrix co-efficient table (行列係数テーブル)
it	image table (画像テーブル)
(B)	この動作に対してオーガナイザをバイパスモードにする
(T)	オペランドがタイルでもよい
-	このオペランドを介してデータの流れがない

【0146】3. 16 データ正規化回路

図49は、3つの主機能ブロックを含むコンピュータグラフィックスプロセッサを示している。3つの主機能ブロックは、ピクセルオーガナイザ246とオペランドオーガナイザB、C247、248中のデータ正規化部1062、主データバス242あるいはJPEG部241の中央グラフィックスエンジン、命令制御部235中のプログラミングエージェント1064である。データ正規化部1062と中央グラフィックスエンジンの動作は、プログラミングエージェント1064への命令ストリーム1064によって決定される。各命令ごとに、プログラミングエージェント1064は復号処理を行い、内部制御信号1067と1068をシステム中の他のブロックに出力する。各入力データワード1069ごとに、正規化部1062は現命令に基づいてデータのフォーマットを行い、処理結果をさらなる処理が実行される中央グラフィックスエンジン1063に送出する。

【0147】データ正規化部は、簡潔にはピクセルオーガナイザとオペランドオーガナイザB、Cを意味する。これらのオーガナイザはデータ正規化回路を含み、入力データを適切に正規化した後、JPEG符号化あるいは主データバス中で中央グラフィックスエンジンに結果を送出する。中央グラフィックスエンジン1063は、32ビットピクセルである標準フォーマットのデータに対して動作する。従って、正規化部は入力データを32ビットピクセルフォーマットに変換する処理を行う。正規化部への入力データワード1069も32ビット幅を有するが、パック要素あるいはアンパックバイトのいずれかのフォーマットであってもよい。パック要素入力ストリームは、データオブジェクトが1, 2, 4, 8, 16

バイト幅であるようなデータワード中での連続するオブジェクトから成る。一方、アンパックバイト入力ストリームは、8ビットのバイトのみが有効であるような32ビットのワードから成る。更に、正規化部で生成されるピクセルデータ11は、チャネルが8ビット幅で定義されるような1, 2, 3, 4個の有効チャネルから成る。

【0148】図50は、データ正規化部1062の具体的なハードウェア構成を示した図である。データ正規化部1062は、FIFOバッファ(FIFO)1073、32ビット入力レジスタ(REG1)、32ビット出力レジスタ(REG2)、正規化マルチプレクサ1075、制御部1076から成る。入力データワード1069はFIFO1073に格納された後、(REG1)1074にすべての入力ビットが所望出力フォーマットに変換されるまでラッチされる。正規化マルチプレクサ1075は、(REG1)1074中の値と(FIFO)1073の現出力とからのビットを選択することで、REG2にラッチされるピクセルを生成するような32組み合わせスイッチを備える。即ち、正規化マルチプレクサ1075は $x[63..32]$ と $x[31..0]$ とで示される2つの32ビット入力ワード1077、1078を入力とする。

【0149】このような手法を用いることで、特に命令処理においてFIFOが少なくとも2つの有効データワードを有する場合に、装置の全体スループットを向上させることができる。これは、データワードをメモリからフェッチする手法によるものである。所望データワードあるいはオブジェクトがFIFOバッファ中の隣接する入力データワードに拡散あるいは「ラップ」されていることがあるが、入力レジスタ1074を用いることで、

FIFOバッファ中の隣接データワードからの要素を用いて完全な入力データを再構成することができ、主データ操作処理段に先立って必要となるさらなる記憶装置やビットストリップ処理を省くことができる。類似のタイプの複数データワードが正規化部に入力されるような場合には、このような構成が大きな利点となる。

【0150】制御部は、REG1 1074やREG2 1076を更新するイネーブル信号REG1_EN1 080やREG2_EN[3..0] 1081を生成するとともに、FIFO 1073や正規化マルチプレクサ1075を制御する信号をも生成する。図49のプログラミングエージェント1064はデータ正規化部1062に対して次のような構成信号を送出する。FIFO_WR4信号、正規化ファクターn[2..0]、ビットオフセットb[2..0]、チャネルカウントc

[1..0]、外部フォーマット(E)といった信号である。入力データは、有効データが存在するクロックサイクルごとにFIFO_WR信号1085を送出することにより、FIFO1073に書き込まれる。領域が得られないときには、FIFOはfifo_full状態フラグ1086を送出する。32ビット入力データが与えられると、外部フォーマット信号を用いて、入力がパックストリームフォーマット(E=1)であるかアンパックバイト(E=0)であるかが調べられる。E=1の場合には、正規化ファクターはパックストリームの各要素サイズとなる。即ち、n=0は1ビット幅の要素、n=1は2ビット幅要素、n=2は4ビット幅要素、n=3は8ビット幅要素、n>3は16ビット幅要素を示す。また、チャネルカウントは、所望有効バイト数でピクセルを生成するためにクロックサイクルごとにフォーマットする連続した入力オブジェクトの最大数である。具体的には、c=1は最小バイトのみが有効であるピクセル、c=2は最小2バイトが有効であるピクセル、c=3は最小3バイトが有効であるピクセル、c=0はすべての4バイトが有効であるピクセルである。

【0151】パックストリームが8ビット幅以下の要素から成る場合には、ビットオフセットがREG1に格納されている値であるx[31..0]中のデータ処理開始位置を決定する。ビットオフセットがはじめの入力バイトの最大ビットからの偏移である場合には、出力データバイトy[7..0]の生成方法は以下の式で与えられる。

n=0の場合、
 $y[i] = x[7-b] \quad 0 \leq i \leq 7 \text{ のとき}$
 n=1の場合、
 $y[i] = x[7-b] \quad i = 1, 3, 5, 7 \text{ のとき}$
 $y[i] = x[6-b] \quad i = 0, 2, 4, 6 \text{ のとき}$
 n=2の場合、

$y[3] = x[7-b]$
 $y[2] = x[6-b]$
 $y[1] = x[5-b]$
 $y[0] = x[4-b]$
 $y[7] = y[3]$
 $y[6] = y[2]$
 $y[5] = y[1]$
 $y[4] = y[0]$
 n=3の場合、
 $y[i] = x[i] \quad 0 \leq i \leq 7 \text{ のとき}$
 n>3の場合、

$y[7..0] = x[15..8]$
 出力データバイトy[15..8]、y[23..16]、y[31..24]を生成する式も同様である。

【0152】なお、以上の手法は、入力ストリームの要素を入力し、必要な回数の複製処理を行い標準幅の出力オブジェクトを生成することで、いかなる長さの出力アレイをも生成することができるように拡張できる。また、入力要素の処理順は、リトルエンディアンでもビッグエンディアンでも良い。なお、上述の例では、常に処理が入力バイトの最大ビットから始まるため、ビッグエンディアン要素順を用いている。リトルエンディアン順を用いる場合には、ビットオフセットを入力バイトの最小ビットに対する値として再定義する必要がある。また、入力要素幅が標準出力幅以上のときには、出力要素は入力要素を切り捨てる、一般には適当な数の最小ビットを削除することによって生成される。上式では、16ビットデータオブジェクトの最大バイトを選択することにより、16ビット入力要素を切り捨てて8ビット幅標準出力を生成している。

【0153】図50の制御部はn[2..0]とc[1..0]の復号を行い、これらとb[2..0]とを用いて正規化マルチプレクサのための選択信号やREG1やREG2のためのイネーブル信号を生成する。また、FIFOは命令中において空になることもあるため、制御部はREG1中に入力データを選択する現在のビット位置in_bit[4..0]と、出力データの書き込みを始める現在のバイト位置out_byte[4..0]を記憶するカウンタを備える。制御部は、処理が終了した時点で、in_bit[4..0]の値とREG1の最終オブジェクトの位置とを比較することで入力ワードを検出し、FIFOが空でない1クロックサイクルにおいてFIFO_RD信号を送出することでFIFO読み出し動作を開始する。信号fifo_empty、fifo_fullはFIFO状態フラグであり、FIFOが有効なデータを有していないときにfifo_empty=1、FIFOがフルのときにfifo_full=1となる。FIFO_RDが送出されたクロックサイクルにおいて、REG1_ENの送出され、新しいデータがREG1に取り込まれる。REG2

のイネーブル信号は、それぞれが出力レジスタの各パイ
トに対応ごとに4つある。制御部は、復号されたc
[1..0]、REG1内の処理待機中の有効要素数、
REG2において未使用チャンネル数の3つの値中の最
小値をとることで、REG2__EN[3..0]を計算
する。E=0の場合には、REG1中には一つの有効要
素しか存在しない。REG2を占めるチャンネル数が復号
されたc[3..0]と等しい場合に、完全な出力ワー
ドが得られる。

$$\begin{aligned} b_trunc[2...0] &= 0 & n \geq 3 \text{ の場合} \\ &= b[2...0] & n = 0 \text{ の場合} \\ &= b[2...1] & n = 1 \text{ の場合} \\ &= b[2] \& "00" & n = 2 \text{ の場合} \end{aligned}$$

(「&」はビットごとの結合処理を示す)

このような処理により、図50においてMUX0、MU
X1...MUX31で示されている各正規化マルチプ
レксаのサイズが、制限機能を用いないときの32-1
からビットオフセット制限を行ったときの最大サイズ2
0-1まで低減される。このサイズ縮小により回路速度
の向上も図ることができる。

【0156】以上のように、好適な実施例では、データ
をいくつかの正規化形式に変換する効率的な回路を備え
る。

3.17 アクセラレータカードの画像処理動作

図2と表2において、命令制御部235はコプロセッサ
224において実行される動作に帰着される命令を「実
行する」。実行される命令は、主データパス部242に
おいて有用な機能が実行されるような種々の命令を含
む。これらの有用な命令の1つが合成処理である。

【0157】3.17.1 合成

図51は、主データパス部242において実装される合
成モデルを示した図である。合成モデル462は、一般
に3つのデータ入力ソースと出力データ(シンク)46
3を含む。入力ソースの1つは、出力463とメモリ内
での同じ相手先からのピクセルデータ464である。ま
た、色や不透明度などのデータソースとして用いられる
命令オペランド465を含む。ここで、色や不透明度は
フラット、ブレンド、ピクセル、タイルのどれでも良
い。なお、フラットやブレンドに関しては、入力/出力

【0154】本発明の好適な実施例では、制御部と正規
化マルチプレксаにおいて用いられるオフセットの一部
のみを用いるなど、ビットオフセットパラメータを制限
する機能を付加することにより、図50の装置が占める
回路領域を大幅に低減することができる。このオフセ
ット制限機能は正規化ファクターに依存するものであり、
以下の式に応じて動作する。

【0155】

を介してフェッチするよりも内部で生成した方が高速で
あるため、ブレンド生成部467において生成される。
更に、入力データは、オペランドデータ465を減衰さ
せる減衰データ466をも含む。

【0158】前述のように、通常ピクセルデータは各チ
ャネルが1バイト幅である4つのチャンネルから成る。こ
こで、最高アドレスの1バイトが不透明チャンネルであ
る。なお、合成処理の動作や有用性に関しては、解説論
文「Thomas Porter and Tom Duff" Compositing Digital I
mages" in Computer Graphics, volume 18, number 3, July
1984」などの標準記事を参照されたい。

【0159】コプロセッサはブレ乗算データを用いるこ
ともできる。ブレ乗算は、各色チャンネルと不透明チャネ
ルとを前もって乗算する処理である。そのため、2つの
オプションのブレ乗算部468、469を備え、必要な
ときに、不透明チャンネル470、471と色データとを
ブレ乗算し、ブレ乗算された出力472、473を得る
ことができる。合成部475は、現在の命令データに基
づいて2つの入力を合成する。以下の表11に、合成オ
ペレータを示す。

【0160】合成動作

【0161】

【表11】

表 11: 合成処理

演算子	定義
$(a_{co}, a_o) \text{ over } (b_{co}, b_o)$	$(a_{co} + b_{co}(1-a_o), a_o + b_o(1-a_o))$
$(a_{co}, a_o) \text{ in } (b_{co}, b_o)$	$(a_{co} b_o, a_o b_o)$
$(a_{co}, a_o) \text{ out } (b_{co}, b_o)$	$(a_{co}(1-b_o), a_o(1-b_o))$
$(a_{co}, a_o) \text{ atop } (b_{co}, b_o)$	$(a_{co} b_o + b_{co}(1-a_o), b_o)$
$(a_{co}, a_o) \text{ xor } (b_{co}, b_o)$	$(a_{co}(1-b_o) + b_{co}(1-a_o), a_o(1-b_o) + b_o(1-a_o))$
$(a_{co}, a_o) \text{ plus } (b_{co}, b_o)$	$(wc(a_{co} + b_{co} - r(a_o + b_o - 255)/255) + r(\text{clamp}(a_o + b_o - 255)/255, \text{clamp}(a_o + b_o)))$
$(a_{co}, a_o) \text{ loadzero } (b_{co}, b_o)$	$(0, 0)$
$(a_{co}, a_o) \text{ loadc } (b_{co}, b_o)$	(b_{co}, a_o)
$(a_{co}, a_o) \text{ loado } (b_{co}, b_o)$	(a_{co}, b_o)
$(a_{co}, a_o) \text{ loadco } (b_{co}, b_o)$	(b_{co}, b_o)

【0162】ここで、 (a_{co}, a_o) は、色 a_c と不透明度 a_o のブレ乗算ピクセルを表す。Rはオフセット値であり、「wc」は以下で説明するラッピング/クランピングオペレータである。なお、上表の各オペレータの逆動作も合成部475が備えていることに注意されたい。クランプ/ラッピング部476は、制限値0～255内にデータをクランプ、或はラップするための処理部である。また、必要であれば、データをオプションの「アンブレ乗算」477処理することもでき、もとのピクセル値に戻すこともできる。最後に、出力データ463が生成され、メモリに戻される。

【0163】図52は、合成処理を行う際に主データバス部に送られる命令形式を示している。主オペコード中のXフィールドが1であれば、前記の表に従って加算オペレータが適用される。このフィールドが0であれば、加算オペレータ以外の他の命令が適用される。Paフィールドは、第一のデータストリーム464（図51）をブレ乗算するかどうかを示すフィールドである。また、Pbフィールドは第2のデータストリーム465をブレ乗算するかどうかを示し、Prフィールドは部位477を用いて結果を「アンブレ乗算」するかどうかを示す。Cフィールドは範囲0～255内にラップあるいはクランプ、オーバーフローあるいはアンダーフローするかどうかを示し、「com-code」フィールドはどのオペレータを適用するかを示す。加算オペレータはオフセットレジスタ(mdp_por)を用いることもできる。このオフセットはラッピング/クランピング処理が行われる前に加算動作の結果から引かれる。加算オペレータでは、com-codeフィールドはオフセットレジスタのチャンネルごとにイネーブルするかどうかを示すフィールドとなる。

【0164】先に述べた図10の標準命令ワード符号化280は、合成オペランドのために変更させられる。出力データの相手先がもとのソースと同じであるため、オペランドAは常に結果ワードと同一となる。そのため、

オペランドAはオペランドBとともにオペランドBをより長く記述することができる。他の命令と同様に、命令中のA記述子は入力フォーマットを記述し、R記述子が出力フォーマットを規定する。

【0165】図53は、ブレンド命令の命令ワードフォーマットを第一例470として示している。ブレンド処理は、各チャンネルごとの開始値471と終了値472とで規定される。同様に、図54は、タイルアドレス476、開始オフセット477、長さ478によって規定されるタイル命令フォーマットを示している。すべてのタイルアドレスやサイズはバイトごとに特定される。タイル処理はモジュラー的に行われ、図55は図54のフィールド476～478を説明する図である。タイルアドレス476はタイルメモリの開始アドレスを、タイル開始オフセット477はタイル開始時に用いられる最初のバイトを、タイル長478はラップする全体のタイル長を指定する。

【0166】図51において、色要素や不透明度は減衰値466によって減衰させられることもある。減衰値は以下の3つの手法により得られる。

1. 命令のオペランドCワード中に減衰ファクタをいれることによって、ソフトウェアがフラット減衰を指定することができる。
2. 1がオンで、0がオフであるビットマップ減衰は、命令のオペランドCワード中でビットマップのアドレスを特定するソフトウェアを用いて利用できる。
3. バイトマップ減衰を、命令のオペランドCワードのバイトマップアドレスに設けてもよい。
4. 定するソフトウェアを用いて、1のときにオン、2のときにオフとするビットマップ減衰を行うことができる。

【0167】減衰値は符号なしの0～255の整数であるため、ブレ乗算された色チャンネルは、

$$Co_a = Co_a \times A / 255$$

を計算することで、減衰ファクターと乗算される。ここ

で、Aは減衰ファクター、C₀はプレ乗算された色チャネルである。

【0168】3. 17. 2 色空間変換命令

図2と表2において、主データバス部242とデータキャッシュ230は、主に色変換の処理を行う。色空間変換は第一の色空間フォーマット（例えば、RGBカラーディスプレイに適したフォーマット）から第二の色空間フォーマット（例えばCYMあるいはCYMK印刷に適したフォーマット）への変換処理を行う。色空間変換処理はすべての色空間をサポートするように設計されており、1次元から多次元までのいかなる機能において用いることができる命令制御部235はCBUS231を介して、主データバス部242、データキャッシュ制御部240、入力インタフェーススイッチ252、ピクセルオーガナイザ246、MUVバッファ250、オペランドオーガナイザB247、オペランドオーガナイザC248、結果オーガナイザ249を構成し、色変換モードで動作するように制御する。このモードでは、ピクセルの複数ラインから成る入力画像がピクセルストリームとして主データバス部242に1ピクセルラインごとに送出される。主データバス部242（図2）は入力インタフェーススイッチ252からピクセルオーガナイザ246を介してピクセルストリームを受け取り、1ピクセルごとに色空間変換処理を行う。また、インターバル表や分数表がMUVバッファ250にあらかじめロードされ、色変換表がデータキャッシュ230にロードされる。主データバス242はこれらの表にオペランドオーガナイザB、Cを介してアクセスし、例えばRGB色空間からCYMあるいはCYMK色空間にピクセルを変換し、変換されたピクセルを結果オーガナイザ249に送る。主データバス部242、データキャッシュ230、データ制御部240、他の前述のデバイスは、命令制御部235の制御のもとで、単一出力一般色空間（SOGCS）変換モードあるいは複数出力一般色空間（MOGCS）変換モードのどちらかのモードで動作する。データキャッシュ制御部240やデータキャッシュ230の詳細に関しては、「データキャッシュ制御部とキャッシュ」240、230（図2）の項目を参照されたい。

【0169】正確な色空間変換処理は複雑な非線形処理である。例えば、RGBピクセルからCYMK色空間の単一主色要素（即ちシアン）への色空間変換処理は理論的には線形であるが、実際には主にピクセルの色要素を出力する出力デバイスにおいて非線形性が生じてしまう。RGBピクセルからCYMK色空間の他の主色要素（黄、マゼンタ、黒）への色空間変換処理においても同様である。即ち、各色要素において生じてしまう非線形性を補償するために、非線形色空間変換が一般に用いられる。このような複雑な色変換処理の非線形性のために、複雑な伝達関数が組み込まれたり、ルックアップテーブルが用いられる。例えば24ビットのRGBピクセル

の入力色空間が与えられると、これらのピクセルをCYMK色空間の8ビット主色要素（シアン）にマッピングするルックアップテーブルは16メガバイト以上を必要とする。同様に、24ビットRGBピクセルをCYMK色空間の4つの8ビット主色要素にマッピングするルックアップテーブルは64メガバイト以上となり、膨大な容量が必要なる。これに対して、主データバス242（図2）は、データキャッシュ230に格納されたルックアップテーブルを用い、入力色空間中の点に粗い出力色値を対応させ、出力色値を補間することで中間出力を得る。

a. 単一出力一般色空間（SOGCS）変換モード
単一ならびに複数出力色変換モード（SOGCS）と（MOGCS）双方とも、RGB色空間は8ビットの赤、緑、青色要素を有する24ビットピクセルから成る。RGB色空間の各RGB次元は15の区間に分割され、それぞれの区間の長さはプリンタのRGBからCYMK色空間への非線形性の逆関数となるように設定される。即ち、伝達関数が高い非線形性を示す場合には区間の長さを短くし、伝達関数が線形に近い場合には区間の長さを長くする。このような伝達関数の非線形部位を知るためには、各出力プリンタの色空間を正確に調べることが望ましい。しかし、ノウハウやプリンタタイプ（例えばインクジェット）の測定された特徴に基づいて、伝達関数を近似あるいはモデル化することも可能である。入力ピクセルの各色チャネルごとに、色要素値の15の区間中の位置が決められる。どの区間に入力色要素値が存在するかを決定するためと、入力色要素値が存在する区間内の位置を決定するためとの2つのテーブルが主データバス部242において用いられる。もちろん、異なる伝達関数を有する出力プリンタに対しては異なるテーブルを用いても良い。

【0170】前述のようにRGBの各次元は15の区間に分割される。即ち、RGB色空間は区間で区切られた3次元ラティス構造となっており、区間の両端の入力ピクセルは入力色空間では粗い配置となっている。更に、区間の両端に対応する出力色空間の出力色値のみがルックアップテーブルに格納されている。従って、入力ピクセルの出力色値は、入力ピクセルが存在する区間の両端に対応する出力色値を決定し、区間内の位置に基づいて出力色値を補間することで求められる。この手法により、大容量のメモリを用いなければならない必要性を低減できる。

【0171】図56は、入力RGB色ピクセルに対して、対応する区間や区間内の位置を決定する例480を示している。変換処理は、24ビット入力ピクセルの8ビット入力色チャネルごとに区間テーブル482や区間内位置テーブル483を用いて実行される。図56において、8ビット入力色要素481は10進数の4をバイナリー形式で表示したものであるが、この8ビット入力

色要素481が区間テーブルや区間内位置テーブルへのルックアップとして用いられる。区間テーブル482は、入力色要素値481が存在する0から14までの区間の1つを4ビットで出力する。同様に、区間内テーブル482は、入力色値要素481が存在する区間内での位置を示す。区間内テーブルは、0から255までの範囲の8ビット値を格納しており、この値は256の分数として解釈される。従って、10進数4をバイナリーであらわした入力色値要素481の場合には、区間テーブル482をルックアップすることで、出力値0が生成される。また、入力値4を区間内位置テーブル483でルックアップすることにより、分数160/256を表わす出力値160が生成される。区間テーブル482と区間内位置テーブル483からわかるように、区間長は均一ではない。前述のように、区間長は伝達関数の非線形性によって決められる。

【0172】上述の通り、各RGB色要素に対して区間テーブルと区間内位置テーブルとを用いることで、3つの区間出力と3つの区間内位置出力が得られる。各色要素に対する区間/区間内位置テーブルはMUVバッファ(図2)にロードされ、必要な時点で主データパス242によってアクセスされる。色変換処理におけるMUVバッファ250の構成を図57に示す。MUVバッファ250(図57)は、それぞれが各色要素に対応する3

$$\begin{aligned} \text{temp11} &= \text{CV}(\text{P0}) + \text{frac_r}(\text{CV}(\text{P1}) - \text{CV}(\text{P0})) \\ \text{temp12} &= \text{CV}(\text{P2}) + \text{frac_r}(\text{CV}(\text{P3}) - \text{CV}(\text{P2})) \\ \text{temp13} &= \text{CV}(\text{P4}) + \text{frac_r}(\text{CV}(\text{P5}) - \text{CV}(\text{P4})) \\ \text{temp14} &= \text{CV}(\text{P6}) + \text{frac_r}(\text{CV}(\text{P7}) - \text{CV}(\text{P6})) \end{aligned}$$

次に、補間処理は、以下の式を用いてtemp21, temp22を求め、緑(G)方向の1次元補間の計算をする。

$$\begin{aligned} \text{temp21} &= \text{temp11} + \text{frac_g}(\text{temp12} - \text{temp11}) \\ \text{temp22} &= \text{temp13} + \text{frac_g}(\text{temp14} - \text{temp13}) \end{aligned}$$

最後に、以下の式に基づいて最終色出力値を求め、青(B)方向の最終次元補間を行う。

$$\text{final} = \text{temp21} + \text{frac_b}(\text{temp22} - \text{temp21})$$

入力と出力との範囲が一致しない場合もしばしば有り得る。ここで、出力範囲が入力範囲よりも狭いと、両端で範囲をクランプしなければならないことが多い。即ち、範囲の端あたりの色を変換した際に望ましくないひずみが生じることが多い。図59は、この問題が生じる例を説明しており、入力範囲値を出力範囲値に1次元マッピングする様子が示されている。ここで、入力値に対する出力値が点510と511とで定まっているものとする。最大の出力値が点512でクランプされるとすると、点511はこの大きさの出力でなければならない。従って、510と511の2つの点を補間する場合に、線515が補間線となり、入力点516には出力値

517が対応する。しかし、範囲の制約が存在しないときに出力値が点518になるような場合には、この手法が必ずしも最適な色マッピングであるとは限らない。510と518との補間線は、入力点516に対して出力値519を生成する。このような2つの出力値517と519の差異は、特に範囲の端あたりの色を印刷する場合などしばしば目につくひずみとなる、この問題を避けるために、主データパス部は、拡張出力色空間で計算し、以下の式に用いて適切な範囲にスケールやクランプすることも可能である。

【0173】図58は、補間処理の例を示した図である。補間処理は、1つの3次元空間500(例えばRGB色空間)から他の色空間(例えばCMYあるいはCMYK)への補間が主な処理である。ピクセルP0からP7はRGB入力色空間内で粗く存在しており、出力色空間において対応する出力色値CV(P0)からCV(P7)を有する。ピクセルP0からP7の間に位置する入力ピクセルPiの出力色要素値は、以下のようにして決定される。まず、入力ピクセルPiを取り囲む区間の両端P0, P1, ..., P7を決定する。次に、区間内位置要素frac_r, frac_g, frac_bを決定し、最後に、P0からP7の両端に対応する出力色値CV(P0)からCV(P7)の間を区間内位置要素を用いて補間する。

【0174】補間処理は、まず赤(R)方向の1次元補間を行い、temp11, temp12, temp13, temp14の値を以下の式から求める。

【0175】

517が対応する。しかし、範囲の制約が存在しないときに出力値が点518になるような場合には、この手法が必ずしも最適な色マッピングであるとは限らない。510と518との補間線は、入力点516に対して出力値519を生成する。このような2つの出力値517と519の差異は、特に範囲の端あたりの色を印刷する場合などしばしば目につくひずみとなる、この問題を避けるために、主データパス部は、拡張出力色空間で計算し、以下の式に用いて適切な範囲にスケールやクランプすることも可能である。

【0176】

$$\text{out} = \begin{cases} 0 & x \leq 63 \text{ のとき} \\ 2(x - 64) & 64 \leq x \leq 191 \text{ のとき} \end{cases}$$

図58において、補間処理は、RGBピクセルを単一出力色要素(例えばシアン)に変換するSOCGS変換モ

ードでも、RGBピクセルをすべての出力色要素に同時に変換するMOGCSモードのどちらでも実行される。色変換が画像中の各ピクセルに対して行われる場合には、数100万ピクセルがそれぞれ独立に色変換されることになる。従って、高速に動作するためには、入力値周辺の8つの値（P0～P7）を素早く見つけることが望ましい。

【0177】図57において説明した通り、主データバス部242は、各色入力チャネルごとに4ビット区間部位と8ビット区間内位置部位とから成る12ビット出力を取り出す。主データバス部242は赤、緑、青色チャネルの4ビット区間部位を結合し、図60中の520のように単一の12ビットアドレス（IR, IG, IB）を生成する。図60は、単一12ビットアドレス520

から単一出力色要素563が得られる様子を示したデータフロー図である。12ビットアドレス520は、まず生成部1881（図141）のようなデータキャッシュ制御部240のアドレス生成部に送られ、メモリバンク（B0, B1, . . . , B7）に対する8個の9ビットライン/バイトアドレス521を生成する。データキャッシュ（図2）は、8個の独立のメモリバンク522に分割され、それぞれは8個のライン/バイトアドレスによって独立にアドレッシングされる。アドレス生成部における12ビットアドレス520から8ライン/バイトアドレスへの変換は、以下の表に従って行われる。

【0178】SOGCSモードにおけるアドレス合成

【0179】

【表12A】

表 12: SOGCS モードのアドレス構成

	Bit [8:6]	Bit [5:3]	Bit [2:0]
Bank 7	R[3:1]	G[3:1]	B[3:1]
Bank 6	R[3:1]	G[3:1]	B[3:1]+B[0]
Bank 5	R[3:1]	G[3:1]+G[0]	B[3:1]
Bank 4	R[3:1]	G[3:1]+G[0]	B[3:1]+B[0]
Bank 3	R[3:1]+R[0]	G[3:1]	B[3:1]
Bank 2	R[3:1]+R[0]	G[3:1]	B[3:1]+B[0]
Bank 1	R[3:1]+R[0]	G[3:1]+G[0]	B[3:1]
Bank 0	R[3:1]+R[0]	G[3:1]+G[0]	B[3:1]+B[0]

【0180】ここで、BIT [8:6], BIT [5:3], BIT [2:0] は、それぞれ9ビットバンクアドレスの6から8ビット、3から5ビット、0から2ビットを示す。また、R [3:1], G [3:1], B [3:1] は12ビットアドレス520の4ビット区間IR, IG, IBの第1から第3ビットまでを示す。表12のメモリバンク5に関して、12ビットから9ビットへのマッピングを詳細に説明する。12ビットアドレス520中の4ビット赤区間Irの1～3ビットが9ビットアドレスB5の6～8ビットにマッピングされ、4ビット緑区間Igの1～3ビットが加算されて9ビットアドレスB5の3～5ビットにマッピングされ、4ビット青区間Ibの1～3ビットが9ビットアドレスB5の0～2ビットにマッピングされる。

【0181】8つのライン/バイトアドレス521は、512×8ビットから成る対応するメモリバンク522へのアドレスとして用いられ、対応する8ビット出力色要素523が各メモリバンク522からラッチされる。このアドレッシング処理によれば、端点P0～P7に対応する出力色値CV（P0）～CV（P7）がメモリバンク中での異なるアドレスとなることがある。例えば、12ビットアドレス0000 0000 0000は、すべてのバンクで000 000 000という同一のバンクアドレスが得られるが、12ビットアドレス0000 000000001の場合には、バンク7、5、3、1ではバンクアドレス000 0000000となり、バンク6、4、2、0ではバンクアドレス000 000

001となるように異なるバンクアドレスが得られる。このようにして、入力ピクセル値を取り囲む8つの単一出力色値CV（P0）～CV（P7）が同時に各メモリバンクから得られ、メモリバンクにおいて出力色値が二重になることを防ぐことができる。

【0182】図61は、単一色変換モードにおいて用いられるデータキャッシュ230のメモリバンクの構成を示している。各メモリバンクは128ラインエントリから成り、各ラインエントリは32ビット長で4×8ビットメモリ533～536から構成される。メモリアドレス521の上7ビットは、メモリアドレス中の対応するデータ列を決定し、メモリバンク出力としてラッチ542するために用いられる。下2ビットはバイトアドレスで、マルチプレクサ543への入力となり、どの4×8ビットエントリを出力として選択544するかを決定するために用いられる。クロックサイクルごとに8つの各メモリバンクのためのデータが出力され、主データバス部242に送られる。即ち、データキャッシュ制御部はオペランドオーガナイザ248（図2）から12ビットのバイトアドレスを受け取り、主データバス部242における補間処理のための8ビット出力色値をオペランドオーガナイザ247、248に出力する。

【0183】図60において、主データバス部242（図2）は補間処理を3ステップで実行する。主データバス部における第1ステップにおいて、乗算/加算部（例えば550）は対応するメモリバンク（例えば522）から出力される色値と赤区間位置要素551を入力

とし、前記の式の第1ステップに従って4つの出力値を計算する。第1ステップの出力（例えば553、554）は第2ステップ556に送られ、`frac_g`入力557を用いて第2ステップの前式に従って出力558を計算する。最後に、第2ステップ出力558、559と`frac_b`入力562とを用いて、前式に基づいて最終出力色563を計算する。

【0184】図60に示した処理は、全体で最大のスループットを得るためにパイプライン化されている。更に、図60の手法は単一出力色要素563が必要なときに用いられる。例えば、図60の手法は、まず出力画像のシアン色要素を生成し、その後でパス間のキャッシュテーブルを再ロードして出力画像のマゼンタ、黄、黒要素を生成するような場合に用いられる。これは、特に、それぞれの出力色が独立パスとなるような4パス印刷処理に適している。

b. 複数出力一般色空間モード

コプロセッサ224はMOGCSモードでの動作も行うが、MOGCSモードはいくつかの点を除いてSOCGSモードとほぼ同様に動作する。MOGCSモードでは、図2の主データパス部242、データキャッシュ制御部240、データキャッシュが協調して、出力される4つの主要要素を同時に出力する。このためにはデータキャッシュ230のサイズが4倍必要となるが、記憶領域を節約するためにMOGCS動作モードでは、データキャッシュ制御部240は出力色空間のすべての出力色値の1/4のみを格納する。出力色空間の残りの出力色値は低速度の外部メモリに格納され、必要な時点で取り出される。なお、本装置や手法は、キャッシュシステムにある粗い色変換テーブルのミス率が非常に小さいという驚くべき事実に基づいている。これは、多くのカラー画像では、1つのピクセルと他のピクセルとの色値の分散が小さいという知見に基づいたものである。また、粗い出力色値は近隣のピクセルにおいても同じになる確率が非常に高い。

【0185】図62は、コプロセッサが複数チャネルキャッシュ色変換を実行する手法を示している。各入力ピクセルは色要素に分解された後、対応する区間テーブル値（図56）が前述のように決定され、`Ir`、`Ig`、`Ib`570といった3つの4ビット区間が得られる。結合された12ビット数570は前述の表12に従って変換され、8個の9ビットアドレスが得られる。アドレス（例えば572）は図63において以下で説明するように再マッピングされ、対応するメモリバンク573をルックアップして4つの色出力チャネル574が得られる。メモリバンク573は、全体で512×32ビットエントリとなり得るが、そのうちの128×32ビットエントリを格納する。メモリバンク573はデータキャッシュ230の一部をなし、図63で説明するようにキャッシュとして用いられる。

【0186】図63は、9ビットバンク入力578が579に再マッピングされる様子を示しており、ビット580～582の順番を入れ替えることによりメモリパターンのエイリアスを取り除くことができる。これにより、隣接するピクセル値が同じキャッシュ要素のエイリアスされる確率を低減することができる。再構成されたメモリアドレス579は、それぞれが32ビットの128エントリから成る対応するメモリバンク（例えば585）へのアドレスとして用いられる。7ビットラインアドレスを用いてメモリ585にアクセスすることで、メモリバンクごとにラッチ586される出力が得られる。各メモリバンク（例えば585）は、それぞれが2ビットの128エントリから成る関連タグメモリを有する。7ビットラインアドレスは、このタグメモリ587中の対応するタグにアクセスするためにも用いられる。アドレス579の最大2ビットをタグメモリ587中の対応するタグと比較することで、出力色値がキャッシュ中に格納されているかどうか決定される。この9ビットアドレス中の最大2ビットは、赤と緑データ区間の最大ビットに対応する（表12参照）。従って、MOGCSモードでは、RGB入力色空間が赤と緑次元において効率よく4象限に分割され、9ビットアドレスの最大2ビットがRGB入力色区間中の象限を指定することになる。即ち、2つのビットタグによって指定された4つの象限に、出力色値が効率的に分割される。このため、あるラインの各タグ値に対応する色出力値は出力色空間で離れて位置することになり、メモリパターンのエイリアスを削減することができる。

【0187】2つのビットタグが一致しない場合には、データキャッシュ制御部はキャッシュミス記録し、必要なメモリ読み出しがキャッシュルックアップ処理とともにデータキャッシュ制御部によって起動される。なお、キャッシュルックアップ処理は、2ビットタグエントリに対応するラインのすべての値が外部メモリから読み出され、キャッシュに格納されるまで停止状態にある。この処理においては、外部メモリに格納されている色変換テーブルの関連ラインを読み出す処理が含まれる。図63の処理575は図62の各メモリバンク（例えば573）ごとに実行されるため、キャッシュ内容によってはメモリバンクから結果（例えば586）が出力されるまでに時間が必要となることもある。データ586の8つの32ビットセットは、この後主データパス部（242）に転送され、上述の補間処理（図62）の3ステップ590～592がすべての色チャネル同時にかつパイプライン処理で実行され、プリンタデバイスに送る4つの色書津力595が生成される。

【0188】実験によれば、一般的な画像におけるキャッシュのミス率が平均で0.01から0.03のピクセルごとのキャッシュラインフェッチであるので、図62と図63において示したキャッシュ機構が有効であるこ

とが示されている。このようなキャッシュ機構を用いることで、多くの場合、データキャッシュ外部のメモリアクセスに対する要求を大幅に低減することができる。

【0189】コプロセッサが行う2つの色空間変換モー

ド(図10)での命令符号化は以下の構造を有する。

色空間変換における命令符号化

【0190】

【表12B】

表 12： 色空間変換におけるインストラクションエンコーディング

オペランド	説明	内部フォーマット	外部フォーマット
Operand A	source pixels	pixels	packed stream
Operand B	multi output channel color conversion tables	other	multi channel csc tables
Operand C	Interval and Fraction Tables	-	I&F table format
Result	pixels	pixels	packed stream
	bytes	unpacked bytes	unpacked bytes, packed stream

【0191】図64は、色空間変換命令における命令フィールド符号化を示したものであり、色変換命令におけるマイナーオペコード符号化は以下になる。

色変換命令におけるマイナーオペコード符号化

【0192】

【表13】

表 13: 色空間変換指示におけるマイナーオペコードエンコーディング

フィールド	説明
trans[3:0]	0 = 変換及びクランプステップをこのチャネルの対応する出力値に適用しない
M	0 = 単一チャネル・カラーテーブルフォーマット 1 = マルチチャネル・カラーテーブルフォーマット

【0193】図65は、MOGCSモードにおいて、RGBピクセルストリームをCYMK色値に変換する手法を示している。ステップS1において、24ビットRGBピクセルストリームがピクセルオーガナイザ246

(図2)に入力される。ステップS2では、図56と図57で説明したように、ピクセルオーガナイザ246がルックアップテーブルを用いて各入力画素の4ビット区間値と8ビット区間内位置とを決定する。入力ピクセルの区間値と区間内位置は、入力ピクセルがどの区間に存在するのか、また区間内のどの位置に存在するのかを表すものである。ステップS3では、主データパス部242が入力ピクセルの赤、緑、青色要素の4ビット区間を結合して、12ビットアドレスワードを生成し、この12ビットアドレスワードをデータキャッシュ制御部240(図2)に送る。ステップS4では、表12と図62において説明したように、データキャッシュ制御部240がこの12ビットアドレスワードを8つの9ビットアドレスに変換する。これらの8つのアドレスは、8つの出力色値CV(P0) - CV(P7)のメモリバンク573(図62)中の位置を示す。ステップS5では、データキャッシュ制御部240(図2)が8つの9ビットアドレスを、図63で説明したように再マッピングする。このようにして、赤と緑の4ビット区間の最大ビットが、9ビットアドレスの最大2ビットにマッピングされ

る。

【0194】ステップS6では、データキャッシュ制御部240が9ビットアドレスの最大2ビットと、メモリ587(図63)中の2ビットタグとを比較する。2ビットタグが9ビットアドレスの最大2ビットと一致しなければ、出力色値CV(P0) - CV(P7)はキャッシュメモリ230に存在しない。従ってステップS7において、2ビットタグエントリに対応する出力色値が外部メモリからデータキャッシュ230に読み込まれる。2ビットタグが9ビットアドレスの最大2ビットと一致する際には、データキャッシュ制御部240はステップS8において図62において説明した要領で8つの出力色値CV(P0) - CV(P7)を取り出す。このようにして、入力ピクセルを取り囲む8つの出力色値CV(P0) - CV(P7)が主データパス部242によってデータキャッシュ230から取り込まれる。ステップS7では、ステップS2で決定された区間内位置を用いて出力色値CV(P0) - CV(P7)が主データパス部242において補間され、補間された出力色値が出力される。

【0195】ここで、RGB色空間や対応する出力色値を4象限以上、例えば32ブロックに更に分割することにより、データキャッシュ容量の格納領域を低減することができることは、専門家にとっては明らかである。3

2ブロックに分割する場合には、データキャッシュの格納容量は出力色値の1/3 2ブロックのみで良い。また、MOGCSモードで用いられるデータキャッシュ機構を単一出力一般変換モードにおいて用いることもできることも、専門家にとっては明らかである。この場合にも、データキャッシュの格納領域を低減することができる。

【0196】3. 17. 3 J P E G符号化／復号
特にメモリの節約やある場所から他の場所への画像転送速度の観点において、画像を符号化して格納することによる利点は計り知れない。画像符号化としてはさまざまな広く流布している標準が生まれている。非常に有名な標準の1つがJ P E G標準であるが、J P E G標準に関する詳細な説明はVan Nostrand Reinholdにより1993年に出版されたPennebakerとMitchellによる著名な本「J P E G : Still Image Data Compression Standard」を参照されたい。コプロセッサ224はJ P E G標準のサブセットを用いて画像を格納する。J P E G標準の利点は、画質を維持したまま大幅な圧縮率が得られる点である。もちろん、画像を圧縮して格納するためには他の標準を用いても良い。J P E G標準は専門家には良く知られた標準であり、A S I C Sに用いることができるようなJ P E Gを実装した種々の製品がJ P E Gコア製品などを含む製造業者から市販されている。

【0197】コプロセッサ224は、1、3、4色要素から成る画像をJ P E G符号化／復号する機能を備えている。1色要素画像はメッシュでもメッシュでなくても良い。即ち、1色要素を、メッシュデータあるいはメッシュされていないデータのどちらかでも取り出すことができる。メッシュデータの例としてピクセルデータごとの3色要素（即ち、ピクセルデータごとのRGB）があり、メッシュされていないデータの例として、画像の各色要素が別々に格納されており各色要素を独立に処理できるようなデータが挙げられる。3色要素画像の場合には、コプロセッサ224は3色チャンネルが最小3バイトに符号化されていると仮定して、ワードごとに1ピクセルを用いる。

【0198】J P E G標準は画像を最小符号化部位（MCU）と呼ばれる小さな2次元部位に分割する。ここで、各最小符号化部位は独立に処理される。J P E G符号化器（図2）は、ダウンサンプリングされた画像の横16ピクセル、縦8ピクセルのMCUでも良いし、ダウンサンプリングされていない画像の場合の横8ピクセル、縦8ピクセルのMCUでも良い。

【0199】図66は、3要素画像をダウンサンプリングする手法を示している。元のピクセルデータ600は、各ピクセルは601がYUV色空間でのY、U、V要素から成るピクセル形式でMUVバッファ250（図

2）に格納されている。このデータはまず4つのデータブロック601～604から成るMCU部位に変換される。データブロックは種々の色要素を含み、ブロック601、602は直接サンプルされたY要素であり、ブロック603、604は図3の例においてサブサンプルされたU、V要素である。ここで、コプロセッサ224は2種類のサブサンプリング機能を備える。1つはフィルタリングしない直接サンプリングであり、奇数のピクセルデータを残し、偶数のピクセルデータを削除するものである。なお、隣接値の平均をとりU、V要素をフィルタリングすることもできる。

【0200】もう一つのJ P E Gサブサンプリングは、図67に示した4色チャンネルサブサンプリングである。このサブサンプリングでは、16×8ピクセル610のピクセルデータブロックが通常のY、U、V要素に加えて不透明度要素（O）を含む4要素611を有している。このピクセルデータ610も図66と同様にサブサンプルされる。しかし、この場合には、不透明チャンネルを用いてデータブロック612、613が作成される。

【0201】図68は、図2のJ P E G符号化器241をより詳細に説明した図である。J P E G符号化／復号器241は、J P E G符号化と復号との双方を行う。符号化処理は、バス620を介してピクセルオーガナイザ246（図2）からブロックデータを受信する。ブロックデータはMUVバッファ250に格納され、ブロックごとに処理がなされる。J P E G符号化処理はいくつかの明確なステップに分割される。これらのステップは、
1. DCT部における離散コサイン変換の実行621
2. DCT出力の量子化622
3. 量子化器622で実行されるジグザグスキャンによるDCT係数の配置
4. 係数符号化器623で実行されるDC DCT係数の予測符号化とAC DCT係数のランレンクス符号化
5. ハフマン符号化器624で実行される係数符号化器の出力の可変長符号化。出力はマルチプレクサ625とRbus626を介して結果オーガナイザ629（図2）に送られる。

【0202】J P E G復号処理は、J P E G符号化動作を逆にしたものである。即ち、J P E G復号処理は、Bus620から圧縮されたJ P E Gブロックを入力する処理を含む。圧縮データはBus630を介してハフマン符号化器624に送られ、データがDC差分とACランレンクスとに復号される。次に、データは係数符号化器623に送られ、ACとDC係数が復号され、通常のスキャンに戻される。その後、量子化器622においてDC係数に対応する量子化値を乗算することでDC係数の逆量子化が行われる。最後に、DCT部621において逆離散コサイン変換が施されもとのデータが復元され、Bus631を介してマルチプレクサ625、Bus626を介して結果オーガナイザに送られる。J P E

G符号化器241は、JPEG符号化器の動作を開始させるために命令制御部によってセットされたレジスタを含むような標準Cbusインタフェース632を介しての通常の方法で動作する。また、量子化器622とハフマン符号化器624はテーブルを必要とするが、これは必要時にデータキャッシュ230からロードされる。テーブルデータは、Obusインタフェース部634を介してアクセスされる。ここでObusインタフェース部634はオペランドオーガナイザB247に接続され、データキャッシュ制御部240と作用しあう。

【0203】DCT部621はピクセルデータに対して離散コサイン変換と逆離散コサイン変換とを行う。DCTに関しては、さまざまな種類のDCT変換実現手法が知られており、「Still Image Data Compression Standard」(同上)の中にも記されているものの、DCT621は以下の項「高速DCT装置」で詳述する高速手法を用いている。なお、DCT変換動作においては、The Transactions of the IEICE, vol. E71, no. 11, November 1988の1095ページに掲載されたAraiらによる論文「Fast DCT-SQ Scheme for Images」に基づくDCT変換手法を用いることもできる。

【0204】量子化器622はDCT係数の量子化と逆量子化を行い、データキャッシュに格納された対応するテーブルから関連値をObusインタフェース部634を介して取り出すことで動作する。量子化処理においては、入力データストリームは、データキャッシュ中の量子化テーブルから読み出された値でもって除算される。この除算は固定小数点の乗算として実装される。また、逆量子化処理では、データストリームは逆量子化テーブル中の値と乗算される。

【0205】図69は、逆量子化622をより詳細に説明した図である。量子化器622は、ローカルバスを介してDCTモジュール621にデータを渡したり、DCTモジュール621からデータを受け取ったりするDCTインタフェース640を備える。量子化処理においては、量子化器622はクロックサイクルごとに2つのDCT係数を受信する。これらの値は量子化器の内部バッファ641、642の1つに書き込まれる。バッファ641、642は入力データをバッファするための2つのポートを備えたバッファである。量子化処理において、DCTサブモジュール621からの係数データはバッファ641、642の1つに格納される。バッファがフルになると、データはバッファからジグザグスキャンで読み出され、Obusインタフェース部634を介して受信した量子化値でもって乗算器643で乗算される。この出力は係数符号化インタフェース645を介して係数符号化器623(図68)に転送される。これらの処理

を行っている間、次のブロックの係数が他のバッファに書き込まれている。JPEG復号処理において、量子化モジュールは、テーブルに格納された値でもって復号されたDCT係数を乗算することで逆量子化処理を行う。量子化と逆量子化とはそれぞれ排他的な動作をするため、乗算器643は量子化と逆量子化との双方において用いられる。なお、逆量子化テーブルへのインデックスとして、 8×8 のブロック中の係数の位置を用いる。

【0206】量子化処理と同様に、2つのバッファ641、642が係数符号化器623(図68)からの入力係数データをバッファするために用いられる。データは量子化値と乗算され、逆ジグザグスキャン順にバッファに書き込まれる。バッファがフルになると、逆量子化された係数が通常の順番でバッファから2つ同時に読み出され、DCTインタフェース640を介してDCTサブモジュール621(図68)に送られる。従って、係数符号化器インタフェースモジュール645は、係数符号化器とのインタフェースとなっており、ローカルバスを介して符号化器にデータを送ったり符号化器からデータを読み出したりする。このモジュールは、符号化時にはジグザグスキャン順でバッファからデータを読み出し、復号時には逆ジグザグスキャン順でバッファにデータを書き込む。DCTインタフェースモジュール640とCbusインタフェースモジュール645ともバッファからの読み出しや書き込みを行うことができる。そのため、アドレス/制御マルチプレクサ647を用いて、各インタフェースがどちらのバッファと動作しているのかを、量子化器のすべてのモジュールを制御するための状態マシンから成る制御モジュール648の制御のもとで、決定する。乗算器643は、 16×8 の2の補数の乗算器を用いてDCT係数を量子化テーブル値と乗算しても良い。

【0207】図68において、係数符号化器623は以下の機能を実行する。

(a) JPEGモードにおけるDC係数の予測符号化/復号

(b) JPEGモードにおけるAC係数のランレンクス符号化/復号

なお、係数符号化器623は、JPEGモード動作とは別に、必要な時点でピクセルの予測符号化/復号やメモリコピー動作のために用いることができると好ましい。係数符号化器623は、ピンクブックに規定されているように、DC/AC係数の予測/ランレンクス符号化/復号を行う。また、JPEG標準に規定されているようなJPEG AC係数のランレンクス符号化/復号に加えて、標準の予測符号化/復号機能も備えている。

【0208】ハフマン符号化器624は、JPEGデータ列のハフマン符号化/復号を行う。ハフマン符号化モードでは、係数符号化器623からランレンクス符号化されたデータが受信され、パックバイトのハフマンスト

リームが生成される。また、ハフマン復号モードでは、ハフマンストリームがPbusインタフェース620からパックバイト形式で読み出され、ハフマン復号された係数が係数符号化モジュール623に送られる。ハフマン符号化器624は、データキャッシュに格納され、Obusインタフェース634を介してアクセスされるハフマンテーブルを利用する。或は、ハフマンテーブルをハードで構成して高速にすることもできる。

【0209】ハフマン符号化においてデータキャッシュを用いるときには、データキャッシュの8つのバンクは、以下に各テーブルごとに詳細に説明されているようにデータテーブルを格納する。

データキャッシュに格納されているハフマン、量子化テーブル

【0210】

【表14】

表 14: データキャッシュに格納されているハフマン、量子化テーブル

バンク	説明
0	バンク0はEHUFCO_DC1あるいはEHUFCOテーブルの256個の16ビットエントリを格納する。インデックスの最小ビットが32ビットワード中の2つの16ビット項目を選択する。このメモリバンクのすべての128ラインが用いられる。
1	バンク1はEHUFCO_DC2テーブルの256個の16ビットエントリを格納する。インデックスの最小ビットが32ビットワード中の2つの16ビット項目を選択する。このメモリバンクのすべての128ラインが用いられる。
2	バンク2はEHUFCO_AC2テーブルの256個の16ビットエントリを格納する。インデックスの最小ビットが32ビットワード中の2つの16ビット項目を選択する。このメモリバンクのすべての128ラインが用いられる。
3	バンク3はEHUFCO_AC2テーブルの256個の16ビットエントリを格納する。インデックスの最小ビットが32ビットワード中の2つの16ビット項目を選択する。このメモリバンクのすべての128ラインが用いられる。
4	バンク4はEHUFSI_DC1あるいはEHUFSIテーブルの256個の4ビットエントリを、EHUFSI_DC2テーブルの256個の4ビットエントリとともに格納する。このメモリバンクのすべての128ラインが用いられる。
5	バンク5はEHUFSI_AC1テーブルの256個の4ビットエントリを、EHUFSI_AC2テーブルの256個の4ビットエントリとともに格納する。このメモリバンクのすべての128ラインが用いられる。
6	未使用
7	バンク7は量子化テーブルの128個の24ビットエントリを格納する。メモリバンクのすべての128ラインの最小3バイトを占める。

【0211】図70において、ハフマン符号化器624は、符号化器660と復号器661との2つの独立のブロックから主に構成される。双方のブロック660、661はマルチプレクサモジュール662を介して同じObusインタフェースを共有する。各ブロックは、それぞれ入力と出力を有し、JPEG符号化器で実行される機能に応じて、一時点ではどちらか1つのブロックのみがアクティブとなる。

a. 符号化

JPEGモードにおける符号化においては、ハフマンテーブルを用いて、DC差分値やACランレンクス値に可変長コード（コードごとに16ビットまで）を割り当てられる。割り当てられたコードは、CCサブモジュールからHCサブモジュールに送られる。また、ハフマンテーブルは動作開始前にデータキャッシュから予めロードされていなければならない。そして、可変長コードをCCサブモジュールから送られてきたDCやAC係数の他のビットと結合し、パックバイト形式が生成される。パック処理の結果、X'FFバイトが得られたとすると、X'00バイトが挿入される。RSTmマーカが必要なときはマーカが挿入されるが、この際には、最後のハフ

マン符号の「1」ビットでのバイト詰込処理と、詰込まれたバイトがX'FFになったときのX'00バイト挿入処理が行われる。RSTmマーカが必要かどうかは、CCサブモジュールによって指示される。また、HCサブモジュールは、Pbus-CCスレーブインタフェース上の「最後の」信号での指示により、画像の最後にEOIマーカを挿入する。EOIマーカの挿入処理においては、RSTmマーカと同様のパック処理、詰込み処理、挿入処理が必要となる。最後に、出力ストリームはパックバイトとして結果オーガナイザ249に送られ、外部メモリに書き込まれる。

【0212】非JPEGモードの場合には、CCサブモジュール（Pbus-CCスレーブインタフェース）からアンパックデータとして符号化器にデータが送られる。各バイトは（JPEGモードと同様に）キャッシュにあらかじめロードされたテーブルを用いて独立に符号化され、可変長シンボルがパックバイト形式にまとめられ、結果オーガナイザ249に送られる。なお、出力ストリームの最後のバイトは1での詰込処理が行われる。

b. 復号

復号アルゴリズムは、高速（リアルタイム）のものと低

速のものとを備える。高速アルゴリズムはJ P E Gモードのみで動作し、低速アルゴリズムはJ P E Gモードでも非J P E Gモードでも動作する。

【0213】高速J P E Gハフマン復号アルゴリズムは、ハフマンシンボルをDC差分値あるいはACランレンクス値のどちらかにマッピングする。これは特にJ P E Gに適するように設計されており、符号化時において例のハフマンテーブル(K3, K4, K5, K6)が用いられることを想定している。なお、これらのテーブルは、キャッシュメモリを参照することなく復号できるように、アルゴリズム中にハード的に埋め込まれている。このような復号処理は、あるデータレートを保証しつつ復号画像を印刷しなければならないような場合を想定したものである。バンド(R S T mマーカで区切られたブロック)を復号するH Cサブモジュールのデータレートは、1クロックサイクルでほぼ1つのD C/A C係数である。H CサブモジュールとC Cサブモジュール間では、データストリームからX' 00挿入バイトを削除するために、1クロックサイクル必要になることもあるが、これはデータに強く依存している。

【0214】ハフマン復号器は高速モードで動作し、クロックサイクルごとに1ハフマンシンボルを抽出する。なお、高速ハフマン復号器については、以下の「可変長符号の復号器」において記している。また、ハフマン復号器661は、ヒープに基づく低速復号アルゴリズムを備えており、図71に示す構造670となっている。

```

    画像の最後までループ
    シンボル長Nを8にセット
    入力ストリームの最初の8ビットをINDEXに格納
    HUFVAL (INDEX) をフェッチ
    I f  HUFVAL (INDEX) == 00xx000111... (ILL)
        「不適切ハフマンシンボル」信号の送出
        e x i t
    e l s e i f  HUFVAL (INDEX) == 1nnn eeee eee
e --- (HIT)
    nnnビットをeeee eeeeに値として転送
    シンボル長N=d e c i m a l (nnn) を転送
    /*000がシンボル長8として*/
    入力ストリームの調整
    b r e a k
    e l s e /*HUFVAL (INDEX) == 01iii iiiiii
i --- (MISS)
    H E A P I N D E X == i i i i i i i i にセット (ヒープベース
を0に仮定)
    N=9にセット
    I f  入力ストリームの第9ビットが0である
        H E A P I N D E X を1増加
        f i
        V A L U E = H E A P (H E A P I N D E X) のフェッチ (第9ビットの
符号)
```

【0215】J P E G符号化ストリームに対して、ストリップー671においてX' 00挿入バイト、X' FF詰込バイト、R S T mマーカが取り除かれ、結合された他のビットとともにハフマンシンボルがシフター672に送られる。なお、ハフマンのみの符号化ストリームではこの処理は行われない。ハフマンシンボル復号の最初のステップは、ハフマンデータストリームの最初の8ビットでアドレッシングされたキャッシュに格納されたH U F V A L テーブルの256のエントリをルックアップする処理である。この値が対応するハフマンシンボルの真の長さである場合には、当該値が出力フォーマッター676に転送され、復号値のシンボル長と付加ビット数とがシフター672にフィードバックされ、関連する付加ビットを出力フォーマッター676に転送し、復号部673に送るハフマンストリームの新しい開始部位を整列する。ここで、付加ビット数は復号値の関数である。最初のルックアップが復号値にならなかった場合、即ちハフマンシンボルが8ビット以上であった場合には、ヒープアドレスが計算され、一致するまで、あるいは「不適切ハフマンシンボル」条件が満たされるまで、引き続きヒープ(キャッシュ内に位置)アクセスが実行される。ルックアップが一致すると上記と同様の処理が行われ、「不適切ハフマンシンボル」条件が満たされた場合にはインタラプト状態となる。

【0216】ヒープに基づく復号アルゴリズムは以下の通りである。

```

Loop
  If VALUE==0001 0000 1111-- (NL)
    「不適切ハフマンシンボル」信号の送出
    exit
  elseif VALUE===1000 eeee eeee
    eeee eeeeを値として転送
    シンボル長Nを転送
    入力ストリームの調整
    break
  else/*VALUE==01iii iii iiii-- (
MISS)
    N=N+1にセット (HEAPINDEX=ii iii i
ii)
    If 入力ストリームの第Nビットが0
      HEAPINDEXを1増加
      fi
    VALUE=HEAP (HEAPINDEX) のフェッチ
    pool
    pool

```

ストリップ671は、入力J PEG 671符号化ストリームからX' 00挿入バイト、X' FF詰込みバイト、RSTmマーカを削除し、「きれいな」ハフマンシンボルを連結された付加ビットとともにシフト672に転送する。ハフマンのみの符号化においては他の付加ビットは存在しないため、このモードにおいては転送されたストリームはハフマンシンボルのみから成る。

【0217】シフト672ブロックは16ビット出力レジスタを備え、次のハフマンシンボルを復号部673に (MSBからLSBの順番のビットストリームで) 転送する。シンボルは16ビット以下であることも多いが、どれだけのビットを解析するかを決定するのは復号部673に任されている。シフト672は復号部673からフィードバック678、即ち現在のシンボル長と (JP

EGモードにおける) 現シンボルに続く付加ビット長とを受信し、シフト672における次のシンボルの開始時点を適切に整列させる。

【0218】復号部673はヒープに基づくアルゴリズムのコアを実装しており、Obus 674経由でデータキャッシュに接続されている。復号部673は、データキャッシュフェッチブロック、ルックアップ値比較部、シンボル長カウンター、ヒープインデックス加算部、付加ビット数の復号部 (復号は復号値に基づいて行われる) を備える。ここで、フェッチアドレスは以下のように解釈される。

【0219】フェッチアドレス

【0220】

【表15】

表 15: フェッチアドレス

フィールド (ビット)	説明
[32:25]	逆量子化(dcquantization)テーブルへの索引
[24:19]	Not used.
[18:9]	ヒープ(heap)への索引
[8:0]	ハフマン復号テーブルの索引

【0221】出力フォーマッターブロック676は8ビット値の復号や (スタンドアロンハフマンモード)、24ビット値と付加ビットとRSTmマーカ情報との32ビットワードへの結合 (J PEGモード) を行う。付加ビットは、復号部673が現シンボルに対する付加ビットの開始位置を決定した後に、シフト672によって出力フォーマッタ676に転送される。また、出力フォーマッタ673は、最終値ワードを予測するために1ワード遅延を用いた2ディープFIFOバッファを備えている。復号処理においては、(高速、低速どちらでも) シ

フト672が入力ビットストリームの最後部の詰込みビットを復号しようと試みるが生じる。このような状態はシフトによって通常検出され、「不適切シンボル」インタラプトを送出する代わりに、「強制終了」信号を送出する。アクティブな「強制終了」信号が送出されると、出力フォーマッタ676は最近の1復号ワード (FIFOにまだ存在している) を「最後」として送出し、復号ストリームに属していない更に最近のワードを削除する。

【0222】図70におけるハフマン符号化器660の

詳細を図72に示す。ハフマン符号化器660はルックアップテーブルを介してバイトデータをハフマンシンボルにマッピングし、符号化部681、シフタ682、出力フォーマッタ683、キャッシュからアクセスされるルックアップテーブルを備える。入力値685はデータキャッシュに格納された符号化テーブルを用いて符号化部681において符号化される。テーブルとしては、符号化すべき値ごとに対応コードを含むテーブルとコード長を含むテーブルとの2つのテーブルが必要となるが、シンボルを符号化する際にはキャッシュ230へのアクセスは一度で良い。なお、JPEG圧縮においては、AC係数とDC係数ごとに別のテーブルが必要となる。また、サブサンプリングが実行されている場合には、サブサンプル要素と非サブサンプル要素ごとに別のテーブルが必要となる。非JPEG圧縮では、2つのテーブル（符号とサイズ）のみが必要である。符号はシフタ682によって処理されて、出力ストリームをビットレベルで構成する。また、シフタ682は、必要時のバイトパディング処理であるRSTmとEOIマーカ挿入処理をも行う。そして、データバイトは出力フォーマッタ683に転送され、X'00バイトでの挿入処理、X'FFバイトやマーカ符号に先立つFFバイトでの詰込処理、

パッキングされたバイトのフォーマット処理を行う。なお、非JPEGモードでは、パッキングされたバイトのフォーマット処理のみが行われる。

【0223】X'FFバイトの挿入処理はシフター682によって行われるため、出力フォーマッタ683はX'FFバイトを前に挿入するために、シフタ682からのどのバイトがマーカであるのかを知る必要がある。これは、バイトに対応しているタグレジスタをシフター682内に備えることによって行われる。バイト境界に存在する各マーカは、マーカ挿入処理においてシフター682によってタグ付けされる。結合処理部683はマーカに先立つX'FF"バイト以降には挿入処理を行わない。タグは、主シフトレジスタと同期してシフトされる。

【0224】ハフマン符号化器はJPEG圧縮において4あるいは8つのテーブルを用い、2つのテーブルを直接ハフマン符号化に用いる。用いるテーブルを以下に示す。

ハフマン符号化器において用いられるテーブル

【0225】

【表16】

表 16: ハフマン符号化器で使用するテーブル

名称	サイズ	説明
EHUFISI	256	Huffman コードサイズ。直接ハフマン符号化で 使用される。符号化された値はインデックス として使用される。
EHUFICO	256	直接ハフマン符号化で使用するハフマン コード値。符号化された値はインデックス として使用される。
EHUFISI_DC_1	16	JPEG 圧縮で DC 係数を符号化するのに使用 されるハフマンコードサイズ。大きさのカ テゴリをインデックスとして使用。
EHUFICO_DC_1	16	JPEG 圧縮で DC 係数を符号化するのに使用 されるハフマンコード値。大きさのカテゴリ をインデックスとして使用。サブサンプ ルされたブロックにも使用される。
EHUFISI_DC_2	16	JPEG 圧縮で DC 係数を符号化するのに使用 されるハフマンコードサイズ。大きさのカ テゴリをインデックスとして使用。サブサ ンプルされたブロックにも使用される。
EHUFICO_DC_2	16	JPEG 圧縮で DC 係数を符号化するのに使用 されるハフマンコードサイズ。大きさのカ テゴリをインデックスとして使用。サブサ ンプルされたブロックにも使用される。
EHUFISI_AC_1	256	JPEG 圧縮で AC 係数を符号化するのに使用 されるハフマンコードサイズ。大きさのカ テゴリとランレングスをインデックスとし て使用。
EHUFICO_AC_1	256	JPEG 圧縮で AC 係数を符号化するのに使用 されるハフマンコードサイズ。大きさのカ テゴリとランレングスをインデックスとし て使用。
EHUFISI_AC_2	256	サブサンプルされ成分の JPEG 圧縮で AC 係数を符号化するのに使用されるハフマン コードサイズ。大きさのカテゴリとランレ ングスをインデックスとして使用。
EHUFICO_AC_2	256	サブサンプルされ成分の JPEG 圧縮で AC 係数を符号化するのに使用されるハフマン コードサイズ。大きさのカテゴリとランレ ングスをインデックスとして使用。

【0226】 3.17.4 テーブルインデックシング
ハフマンテーブルは、コプロセッサデータキャッシュ 2
30において局所的に格納されている。データキャッ
シュ 230は、各ラインが8ワードから成る128ライン
の直接マッピングキャッシュとして構成される。キャッ
シュライン中の各ワードは独立にアドレスすることがで
き、この特徴をハフマン復号器が利用して同時に複数の
テーブルにアクセスする。テーブルは小さい(≤256
項目)なので、Obusの32ビットアドレスフィールド

で複数のテーブルへのインデックスを含めることがで
きる。

【0227】 上述のように、JPEG低速復号モードで
は、様々なハフマンテーブルを格納するためにデータキ
ャッシュが用いられる。データキャッシュのフォーマッ
トを以下に示す。

ハフマン/量子化テーブルのバンクアドレス

【0228】

【表17】

表 17: ハフマン/量子化テーブルのバンクアドレス

バンク	説明
0~3	1024 個、16 ビットのヒープ項目を保持する。最小インデックスビットにより各バンク中の2つの 16 ビットワードを選択する。4つのメモリバンクのすべての 128 ラインが用いられる。
4	DC ハフマン復号テーブルの 12 ビット項目のうち最小8ビットを 512 個保持する。インデックス中の最小2ビットにより、32 ビットワード中の4つのバイト項目を選択する。メモリバンクのすべての 128 ラインが用いられる。
5	AC ハフマン復号テーブルの 12 ビット項目のうち最小8ビットを 512 個保持する。インデックス中の最小2ビットにより、32 ビットワード中の4つのバイト項目を選択する。メモリバンクのすべての 128 ラインが用いられる。
6	DC ならびに AC ハフマン復号テーブルの最大4ビットを保持する。各インデックスの最小2ビットにより、各ワード中の4つのニブルを選択する。
7	量子化テーブルの 24 ビット項目を 128 個保持する。メモリバンクのすべての 128 ラインの最小3バイトを占める。

【0229】JPEG符号化器241（図2）においてJPEG命令が実行されるのに先立ち、画像次元レジスタ（PO_IDR）あるいは（RO_IDR）に適切な画像幅値がセットされなければならない。他の命令とともに、命令の長さは処理すべき入力データ項目数に関係する。これはいかなるパディングデータをも含み、用いられているサブサンプリングオプションや色チャンネル数にも関連する。

【0230】コプロセッサ224により出されたすべての命令は、生成する出力データ量を制限するために2つの機能を用いる。これらの機能は、入力と出力データのサイズが異なるときにもっとも有効であり、特にJPEG符号化/復号のように出力データサイズが未知であるときに有効である。これらの機能は、出力データを書き出すか、命令が適切に実行されたように見せながら単にデータを削除するかを決定する。デフォルトではこの機能はオフになっており、RO_CFGレジスタ中の適切なビットをイネーブルにすることでオンとなる。しかし、JPEG命令ではこのビットをセットする特別なオプションが用意されている。なお、JPEG圧縮を用いる際には、コプロセッサ224は出力データの「削除」や「制限」機能をサポートすることが望ましい。

【0231】図73を用いて、削除、制限処理を説明する。入力画像690は、ある高さ691とある幅692とを有する。ここで、画像の一部分のみに関心があり、他の部位は印刷するのには関係がないというような状況がしばしば存在する。しかしながら、JPEG符号化シ

ステムでは8×8ピクセルブロックを対象とする。そのため、画像の幅が8の倍数とならない場合や、MCU695を構成する関心部位領域がきちんと境界と一致しない場合が生じる。そこで、出力削除レジスタRO_CUTは、出力データストリームのはじめの部位696において削除する出力バイト数を決定する。また、出力制限レジスタRO_LMTは、生成する最大出力バイト数を決定する。この最大出力バイト数は、削除レジスタの結果に基づいてメモリに書込まれないバイトをも含む。このような処理により、最終出力バイト698以降のデータは出力されないような最終出力バイトを求めることができる。

【0232】JPEG復号器における削除、制限機能が特に有効であるケースとして2つの場合がある。第1のケースは、図74に示すように、復号画像の1ストリップ701の一部位700を抽出あるいは解凍する場合である。第2のケースは、図75に示すように、全体の画像714において、複数の完全なストリップ（例えば、711、712、713）の抽出あるいは解凍が必要となる場合である。

【0233】JPEG命令の命令フォーマットやフィールド符号化を図76に示す。マイナーオブコードフィールドの説明を以下に記す。

命令ワード—マイナーオブコードフィールド

【0234】

【表18】

表 18: 命令ワード:マイナーオPCODEフィールド

フィールド	説明
D	0= 符号化 (圧縮) 1= 復号化 (伸長)
M	0= 単一カラーチャネル 1= マルチチャネル
4	0= 3チャネル 1= 4チャネル
S	0= サブ/アップ・サンプリング様式を使用しない 1= サブサンプリング様式を使用する
H	0= 高速ハフマン符号化を使用する 1= 汎用のハフマン符号化を使用する
C	0= カットレジスタ使用しない 1= カットレジスタを使用する
T	0= 出力を切り捨てない 1= 出力を切り捨てる
F	0= サブサンプリングの前に低域フィルタを用いない 1= サブサンプリングの前に低域フィルタを用いる

【0235】3. 17. 5 データ符号化命令

コプロセッサ224は図2のJPEG符号化器の一部を他の用途で用いることができる機能を備えることが望ましい。例えば、ハフマン符号化はJPEGのみならず他の圧縮手法においても用いられる。また、階層的画像復号のためのみにハフマン符号化部を制御するデータ符号化命令が備わっていることも望ましい。更に、ランレンクス符号化器/復号器、予測符号化器も同様の命令でもって独立に用いられることができる。

【0236】3. 17. 6 高速DCT装置

従来の図77に示したような離散コサイン変換(DCT)装置では、まず8×8ブロックの列方向に対して1次元DCTを実行し、次いで8×8ピクセルブロックの行方向に更に1次元DCTすることにより、8×8ピクセルブロックの2次元変換を実行する。このような装置では、入力回路1096、算術回路1104、制御回路1098、置換メモリ回路1090、出力回路1092を一般に備える。

【0237】入力回路1096は8×8ブロックから8ビットピクセルを受信する。入力回路1096は、中間マルチプレクサ1100、1102を介して算術回路1104に接続されている。算術回路1104は、8×8ブロックの完全な列あるいは行に対して算術処理を行う。制御回路1098は、他の全ての回路を制御し、DCTアルゴリズムを実行する。算術回路の出力は、置換メモリ1090、レジスタ1095、出力回路1092に送られる。置換メモリは更にマルチプレクサ1100に接続され、マルチプレクサ1100は次のマルチプレクサ1102に出力を送出する。また、マルチプレクサ1102はレジスタ1094からのデータをも受信する。置換回路1090は8×8ブロックデータを列形式で入力し、行形式でデータを出力する。出力回路1092はピクセルデータの8×8ブロックに対するDCT係数を出力する。

【0238】通常のDCT装置では、算術回路1104がもっとも複雑であるため、算術回路1104の速度が全体の装置速度を決定する。図77の算術回路1104

は、一般に算術処理を図78を用いて説明するように複数の処理段階に分割して処理を行う。従って、各処理段階1144、1148、1152、1156を加算器や乗算器などの通常の資源を用いて実行するような単一回路が用いられる。このような算術回路1104では、単一の共通回路が回路1104の種々の処理段階を実行するために用いられるため、最適速度に比べて速度が遅くなるという欠点を有する。また、中間結果を蓄える格納手段もこれに含まれる。回路のクロックサイクル時間は少なくとも最も遅い回路段階以上でなければならないため、全体の処理に要する時間は各処理段階に要する時間の和以上となり得る。

【0239】図78は、図77の装置における通常の算術データパスを示したものであり、DCTを4処理段階で行う処理の一部を示している。なお、本図は実際の実装を示したものでなく、機能を示したものである。4処理段階1144、1148、1152、1156のそれぞれは、単一の再構成可能な回路として構築される。サイクルごとに、1次元DCTの4処理段階1144、1148、1152、1156のそれぞれが再構成される。また、この回路においては、4処理段階1144、1148、1152、1156のそれぞれが共通の資源(加算器や乗算器など)のプールを用いることで、ハードウェア規模を小さくしてえる。

【0240】しかしながら、この回路の欠点は速度が最適になっていないことである。4処理段階1144、1148、1152、1156はそれぞれが加算器や乗算器の同一プールから構成されている。そのため、クロックピリオドは最も遅い処理段階によって決定される(この例ではブロック1144の20ns)。入力と出力マルチプレクサ1146と1154の遅延(それぞれ2ns)と、フリップフロップ1150の遅延(3ns)を足すと、全体の遅延が27nsとなる。従って、このDCT構成では最速27nsで動作する。

【0241】パイプライン形式のDCT構成もよく知られている。この構成の欠点は、多量のハードウェアを必要とする点である。スループットの観点では本発明の構

成ではパイプライン構成に及ばないものの、現在のほとんどのDCT構成と比べてきわめて良好な性能／サイズ特性や速度特性を示す。図79は、ピクセルデータが入力回路1126に入力され、8ビットピクセルデータの列を格納するようなJPEG符号化器(図2)において用いられる好適な離散コサイン変換部の構成を示した図である。置換メモリは、2次元離散コサイン変換の2回目のパスを実施するために、列形式データを行形式データに変換する。入力回路1126と置換メモリ1118からのメモリは、マルチプレクサ1124においてマルチプレキシングされ、出力データが算術回路1122に送られる。算術回路1122の結果は、2回目のパスの終了後出力回路1120に送られる。制御回路1116は、離散コサイン変換装置中のデータの流れを制御する。

【0242】離散コサイン変換処理の第1回目のパスでは、変換すべき画像の列データあるいはピクセルデータに逆変換される変換画像係数が、入力回路1126に送られる。このパスでは、マルチプレクサ1124は制御回路1116によって設定され、入力回路1126から算術回路1122にデータが送られる。図80は、算術回路1122の構成をより詳細に示した図である。フォワード離散コサイン変換の実行の場合には、フォワード離散コサイン変換を実行するフォワード回路1138の結果がマルチプレクサ1124において選択される。ここで、マルチプレクサ1124は制御回路1116によって設定される。逆離散コサイン変換の実行の場合には、制御回路1126の設定に基づいて、逆回路1140からの出力がマルチプレクサ1142において選択される。1回目のパスでは、各列ベクトルが算術回路1122(制御回路1116によって適切に設定される)によって処理された後、当該ベクトルが置換メモリ1118に書込まれる。8×8ブロック中のすべての8列ベクトルの処理が終わり、置換メモリ1118に書込まれると、離散コサイン変換の2回目のパスが開始される。

【0243】フォワードあるいは逆離散コサイン変換の2回目のパスでは、行形式のベクトルが置換メモリ1118から読み出され、マルチプレクサ1124を介して算術回路1122に送られる。このパスでは、マルチプレクサ1124は入力回路1136からのデータを無視し、置換メモリ1118からの行ベクトルデータを算術回路1122に転送するように、制御回路によって設定される。算術回路1122中のマルチプレクサ1142は、逆回路1140からの結果データを算術回路1122の出力に送る。算術回路1122からの結果が得られた時点で、制御回路1116からの指令に基づいて出力回路1120は結果を取り込み、以降の時点で出力する。

【0244】算術回路1122は、中間結果を格納する記憶部位を持たないという点で、組み合わせ回路となっ

ている。制御回路1116は、データが入力回路1136からマルチプレクサ1124や算術回路1122を介して出力されるまでに要する時間を把握しているため、算術回路1122の出力からの結果ベクトルを出力回路1120に取り込む時点を正確に指示することができる。算術回路1122において中間記憶を持たない利点は、中間記憶要素との間でのデータのやり取りに必要な時間を省くことができるとともに、算術回路1122をデータが通過するのに要する時間が内部処理段すべての和となり、最大の時間を要する処理段のN倍(従来の離散コサイン変換装置のように)にはならないことが挙げられる。なお、ここで、Nは算術回路中の処理段数である。

【0245】図81は、全体の遅延が単に4つの処理段1158、1160、1162、1164の和、 $20ns + 10ns + 12ns + 15ns = 57ns$ となり、図78の回路よりも高速となることを示している。このような回路によれば、全体のシステムクロックサイクルを短くすることができる。図81の回路において、結果を得るのに4クロックサイクルが必要であるとする、全体のDCTシステムにおいて最小実行時間は $57/4ns$ (14.25ns)となり、図78ではDCTクロックサイクルが27nsとせざるを得ないことを鑑みると大幅な性能向上となることがわかる。

【0246】本DCT装置の実際の実行時においては、Yukihiro Arai, Takeshi Agui, Masayuki NakajimaらによるThe Transactions of the IEICE, vol. E71, no. 11, 1988年11月のページ1095に掲載された論文「画像のための高速DCT-SQ手法」で示されたDCTアルゴリズムを用いることもできる。このアルゴリズムをハードウェアで実行することで、本DCT装置中の算術回路1122に容易に配置することができる。同様に、他のDCTアルゴリズムを算術回路1122中にハードウェアとして配置することも可能である。

【0247】3. 17. 7: ハフマン復号器

以下の実施例は、種々の長さのビットフィールドがインターリーブされた可変長符号に対する手法と装置に関するものである。特に、本発明の実施例は、可変長符号化データの効率の良い、高速な、単一処理段(クロックサイクル)の復号を提供するものである。ここで、可変長符号化されていず整列されているようなデータとは、既に別の前処理ブロックにおいて符号化データストリームから削除されているものとする。更に、削除されたバイト整列データの位置情報は、復号されるデータと同時に復号器の出力に送られる。また、前処理された入力データ中に残っているバイト整列、非可変長符号化ビットフィールドの高速な検出、並びに削除をも提供するものである。

【0248】本発明の好適な実施例では、マーカ符号間のクロックサイクルごとに1ハフマンシンボルといったレートで、J P E G符号化データを復号することのできる高速ハフマン復号器を備えることが望ましい。これは、別の前処理ブロックにおいて、入力データからバイト整列されハフマン符号化されていないマーカヘッダ、マーカ符号、挿入バイトを分離し、除去する手法によって実現できる。バイト整列されたデータが除去されると、入力データはデータシフト組み合わせ回路ブロックに送られ、データ復号レジスタの連続的な挿入処理を行い、復号部位にデータが送られる。もとの入力データから除去されたマーカの位置はマーカシフトブロックに送られ、データシフトブロックにおいてシフトされた入力データと同時にマーカ位置ビットのシフトが行われる。

【0249】復号部は、データ復号レジスタから入力された符号化ビットフィールドを組合せ回路で復号する。復号部の出力は、復号値(v)と入力符号の実際の長さ(m)である。ここで、mはn以下である。また、可変長ビットフィールドの長さ(a)も出力する。ここで、aは0以上の値である。可変長ビットフィールドはハフマン符号化されていないため、すぐにハフマン符号化される。復号部の入力中の長さnのビットフィールドは実際の符号以上の長さを有する。復号部では、実際のコード長(m)を決定し、他のビット(a)の長さとともに制御ブロックに転送する。制御ブロックはシフト値(a+m)を決定し、データ/マーカシフトブロックを起動して次の復号サイクルに備えて入力データをシフトする。

【0250】本発明の装置では、復号値、入力符号の実際の長さ、ハフマン符号化されていないビットフィールドの長さを所定の時間内に出力するものであれば、ROM, RAM, P L Aなどのいかなる組合せ回路の復号部を用いることができる。本実施例では、復号部は、J P E G標準で規定されているように予測符号化D C係数値やA Cランレンクス値を出力する。また、J P E G標準で規定されているように、復号値と同時に入力データから除去されたハフマン符号化されていないビットフィールドは、D CとA C係数の値を決定する付加ビットを示す。データ復号レジスタ中のデータから除去されたハフマン符号化されていないビットフィールドの他の種別としては、J P E G標準に規定されているようにもとの入力データストリーム中のバイト整列マーカに先立つパディングビットがある。これらのビットは、制御ブロックがデータレジスタのパディング領域の内容をチェックすることによって検出される。パディング領域はデータレジスタのk最大ビットから成り、マーカレジスタの最大ビット中のマーカビットの存在によって示される。パディング領域中のすべてのビットが同一(J P E G標準では1)であれば、パディングビットとして判断され、復号されることなくデータレジスタから除去される。そし

て、次の復号サイクルに向けて、データとマーカレジスタの内容は更新される。

【0251】装置の実施例では、本発明の好適な実施例の要求に応じて、出力データのフォーマット処理を行う出力ブロックを備える。出力ブロックは、J P E Gにおける付加ビットなどのように、対応する可変長符号化されていないビットフィールドや、J P E Gにおけるマーカのように整列された入力バイトや符号化されていないビットフィールドの位置を示す信号とともに、復号値を出力する。

【0252】J P E G符号化器241(図2)によって復号されたデータは、J P E Gコンパチブルであり、「付加ビット」と呼ばれる可変長符号化されていないビットフィールド、「パディングフィールド」と呼ばれる可変長符号化されていないビットフィールド、「マーカ」「挿入バイト」「詰込バイト」と呼ばれる固定長の、バイト整列された、符号化されていないビットフィールドがインターリーブされた可変長ハフマン符号化コードから構成される。図82に代表的な入力データを示す。

【0253】J P E G符号化器241のハフマン復号器中の全体構成やデータフローを図83と図84に示す。図83は、J P E Gデータのハフマン復号器の構成を詳細に示している。ストリップ1171はマーカ符号(符号F F X X h e x, X Xは非零)を除去し、バイト(符号F F h e x)を挿入し、バイト(符号F f h e xに続く符号0 0 h e x)を詰込む。これらはすべて入力データのバイト整列された要素であり、32ビットワードとしてストリップに送られる。処理すべき第1ワードの最大ビットは、入力ビットストリームの先頭になる。ストリップ1171では、バイト整列されたビットフィールドが、ハフマン符号の復号処理が復号器のダウンストリーム部位において実際に行われる前に、入力データから除去される。

【0254】入力データはストリップ1171にクロックサイクルに1つごとの32ビットワードとして入力される。入力バイト1211を0から3への番号付けを図85に示す。番号(i)のバイトが挿入バイト、詰込バイト、あるいはマーカであるため除去されたとすると、番号(i-1)から0の残りのバイトがストリップ1171の出力で左にシフトされ、番号(i)を1減らす。この際、バイト0は「無関係な」バイトとなる。ストリップ1171から出力されたバイトの有効性は、図85に示されている別の出力タグ1212によって符号化される。ストリップ1171によって除去されないバイトはストリップにおいて左詰めで出力される。出力中の各バイトは、対応するバイトが有効(ストリップ1171を通過する)か、無効(ストリップ1171で除去される)か、有効かつマーカの後部か、を示すタグが付加される。タグ1212は、データシフタを通してデータレ

ジスタ1182へのデータバイトのロードを制御するとともに、マーカーシフトを通してマーカーレジスタ1183へのマーカー位置のロードを制御する。入力ワードから1バイト以上削除された場合でも同様の手法が実行される。すなわち、すべての残りの有効バイトが左詰めされ、対応する出力タグが出力バイトの有効性を示す。図85には、種々の入力バイトの組み合わせに対する出力バイトと出力タグの例1213が示されている。

【0255】図83において、プレシフトとポストシフトブロック1172、1173、1180、1181の役割は、データレジスタ1182とマーカーレジスタ1183に十分な空き領域がある場合にデータレジスタとマーカーレジスタとに連続的にデータをロードすることである。データシフトとマーカーシフトブロックは、プレシフトブロックとポストシフトブロックとから成るが、それぞれは同一であり同様に制御される。差異は、データシフトがストリップ1171からのデータを処理するのに対し、マーカーシフトはタグのみを処理し、マーカー位置を復号されたハフマン値と同時に復号器に出力する点にある。ポストシフト1180、1181の出力は、図83に示されているように対応するレジスタ1182、1183に直接転送される。

【0256】図86にもデータプレシフト1172が示されているが、データプレシフト1172は、ストリップ1171からのデータに32個のゼロを最小ビット1251に付加し、64ビットにデータを拡張する。次いで、拡張データは64ビット幅のパレルシフト1252で右にデータレジスタ1182に現在存在するビット数だけシフトされる。この際、ビット数は、データ1182、マーカー1183レジスタ内にどれだけの有効ビットが存在するかを常に把握している制御ロジック1185から与えられる。そして、パレルレジスタ1252は、64ビットを、64個の2×1基本マルチプレクサ1254から成るマルチプレクサブロック1253に転送する。各基本2×1マルチプレクサ1254は、パレルシフト1252からの1ビットとデータレジスタ1182からの1ビットを入力とする。データレジスタ中のビットが有効であるときにデータレジスタビットを出力する。一方、無効である場合には、パレルシフト1252のビットを出力する。すべての基本マルチプレクサ1254への制御信号は、図86ならびに図87におけるレジスタ1223のプレシフト制御ビット0...5として示されているように制御ブロックのシフト制御1信号より復号される。基本マルチプレクサ1254の出力はパレルシフト1255に送られ、図86に示されるように5ビット制御信号シフト制御2より与えられるビット数分左にシフトされる。これらのビットは、データレジスタ1182において現データの復号によって使用されるビット数を示したものであり、現復号ハフマンコード長と続く付加ビット数、あるいはパディングビットが検

出されていれば削除されるパディングビット数、あるいはデータレジスタ1182中の有効ビット数が削除されるビット数以下であれば0を足したものとなる。このようにして、パレルシフト1255から出力されるデータには、単一復号サイクルの後にデータレジスタ1182にロードされる新しいデータが含まれることになる。データレジスタ1182の内容は、最大ビットが復号されるためにレジスタからシフトアウトされ、ストリップ1171から0、8、16、24、32ビットがデータレジスタ1182に付加されるといった具合に変更される。データレジスタ1182に復号できるだけの十分なビットが存在しない場合には、ストリップ1171からのデータが存在すれば現サイクルにおいてロードされる。現サイクルにおいてストリップ1171からのデータが存在しない場合には、データレジスタ1182からの復号ビットは、十分なビット数であれば削除され、十分なビット数でなければデータレジスタ1182の内容は変更されない。

【0257】マーカープレシフト1173、ポストシフト1181、マーカーレジスタ1183は、データプレシフト1172、データポストシフト1180、データレジスタ1182とそれぞれ同一の部位である。部位1173、1181、1183内のデータフローならびにこれらの部位間のデータフローも、部位1172、1180、1182間でのデータフローと同一である。同様の制御信号が制御部1185より双方の部位セットに送られる。これらの部位の差異は、マーカープレシフト1173とデータプレシフト1172の入力データ種別と、マーカーレジスタ1183とデータレジスタ1182の内容がどのように用いられるか、という点である。図88に示すように、ストリップ1171からのタグ1261は8ビットワードとして入力され、データレジスタ1182に向かうデータバイトごとに2ビット割り当てられている。図85に示した符号化手法によれば、有効かつマーカー後部であるバイトを示す2ビットタグの最大ビットは1である。ストリップ1171から同時に送られる4つのタグの最大ビット位置のみが、マーカープレシフト1173の入力1262として送出される。このようにして、マーカープレシフトへの入力には、はじめに符号化されたデータビットでマーカーの後部に位置する位置を示す1がセットされたビットが存在することになる。同時に、これらはデータレジスタ1182中でマーカーが後に続くはじめに符号化されたデータビットの位置をマークしている。マーカーレジスタ1183中のマーカー位置ビットとデータレジスタ1182中のデータビットの同期的な振る舞いによって、制御ブロック1185はパディングビットの検出や削除を行うことができるとともに、復号データと同時にマーカー位置を復号器の出力に送出することができる。上述の通り、2つのプレシフト（データ1172とマーカー1173）、ポストシフト（データ1

180とマーカ1181)、レジスタ(データ1182とマーカ1183)は同一の制御信号を与えられているため、完全な並列、同期動作が可能となる。

【0258】復号部1184(図89にも示されている)は、データレジスタ1182の最大16ビットを入力し、復号されたハフマン値、復号される現在の入力符号長、入力符号に続く付加ビット長(復号値の関数となる)を抽出するための組み合わせ回路復号部1184に送られる。付加ビット長は、対応する前のハフマンシンボルが復号された時点で明らかになり、次のハフマンシンボルの開始位置となる。従って、クロックサイクルごとに1つの値が復号される速度を維持する場合には、ハフマン値の復号を組み合わせ回路ブロックで行わなければならない。復号部は、図89に示すように、16ビットトークンをデータレジスタ1182から入力し、ハフマン値(8ビット)、対応するハフマン符号化されたシンボル(4ビット)、付加ビット(4ビット)を生成するような組み合わせ回路ブロックとしてハードワイヤされた4つのPLAスタイルの復号テーブルを備えることが望ましい。

【0259】パディングビットの削除処理は、制御部1185の一部であるパディングビットの復号部においてデータレジスタ1182中でパディングビット列が検出された際の実際の復号処理において行われる。図90にパディングビットの復号部を示す。マーカレジスタ1183、1242の8最大ビット中にマーカ位置ビットが存在するかどうか調べられる。マーカ位置ビットが存在した場合には、マーカレジスタ1242中のマーカビットに先立つビットに対応するデータレジスタ1182、1241中のすべてのビットが現在のパディング領域として判断される。現在のパディング領域の内容は、パディングビット検出部1243によってすべて1であるかどうかチェックされる。現パディング領域のすべてのビットが1である場合には、パディングビットであると判断されデータレジスタから削除される。ここで、削除処理は、データレジスタ1182、1241(同時にマーカレジスタ1183、1242)の内容を対応するシフト1172、1173、1180、1181を用いて1クロックサイクルで左にシフトさせることで行われる。この処理は、復号値が出力されないことを除いて通常の復号モードと同一である。現パディング領域のすべてのビットが1でない場合には、パディングビット削除サイクルではなく通常の復号サイクルが実行される。パディングビットの検出は上述のように各サイクルごとに行われ、データレジスタ1182にパディングビットが存在する場合には削除される。

【0260】図87は、制御部1185を詳細に示したものである。制御部の中心部位はレジスタ1223であり、データレジスタ1182中の現有効ビット数を保持している。マーカレジスタ1183中の有効ビット数は

常にデータレジスタ1182中の有効ビット数と等しい。制御部は3つの機能を実行する。第一の機能は、レジスタ1223に格納されるデータレジスタ1182中の新しいビット数の計算である。第二の機能は、シフト1172、1173、1180、1181、1186、1187、復号部1184、出力フォーマット部1188への制御信号の生成である。第三の機能は、上述のようにデータレジスタ1182中のパディングビットの検出である。

【0261】データレジスタ1182中の新しいビット数(new_nob)は、データレジスタ1182(nob)中の現ビット数と現サイクルにおいてストリップ1171からロード可能なビット数(nos)との加算し、現サイクルにおいてデータレジスタ1182から削除されるビット数(nor)を減算したものと計算される。ここで、現サイクルは、復号サイクルあるいはパディングビット削除サイクルである。従って、新しいビット数は以下のように計算される。

【0262】 $new_nob = nob + nos - nor$
これらの処理は加算器1221と減算器1222とで実行される。なお、現サイクルにおいてストリップ1171からデータが入力されない場合には(nos)が0となる。また、データレジスタ1182においてビットが足りない、即ちデータレジスタ中のビットが制御部1185からの現符号長と続く付加ビット長との和以下であることにより、現サイクルにおいて復号処理が行われない場合にも(nos)は0となる。値(new_nob)は64を越えることがあり、ブロック1224において越えているかどうかチェックされる。このような場合には、ストリップ1171は停止状態となり、新しいデータのロードがなされない。マルチプレクサ1233は、ストリップ1171からロードされたビット数をゼロにするために用いられる。ここで、ストリップ1171を停止させる信号は図示されていない。復号部1231からの信号「パディングサイクル」はマルチプレクサ1234を制御し、パディングビット数あるいは復号ビット数(符号ビットと付加ビットとの長さ)を削除すべきビット数(nor)として選択する。復号ビット数がデータレジスタ中のビット数(nob)以上であると、比較器1228において判断されると、マルチプレクサ1234に与えられるシフトすべき有効ビット数はNANDゲート1230においてゼロに設定される。すなわち、(nor)はゼロに設定され、データレジスタのビットの削除は行われない。マルチプレクサ1234の出力は、ポストシフト1182と1183の制御にも用いられる。データレジスタ1182の幅はデッドロック状態を避けるように設定される。すなわち、ストリップ1171からの最大ビット数を収容するだけの領域をデータレジスタに確保するように、あるいは復号/パディングビット削除サイクルの結果として十分な有効ビッ

ト数が削除されるように設定される。

【0263】復号サイクルにおいて削除されるビット数の計算は加算器1226において実行される。オペランドは組み合わせ回路復号部1184から入力される。16ビットの符号長は復号部において“0000”と符号化されるため、“ou_reduce”ロジック1225では“0000”が“10000”に符号化され、現在の符号なしのオペランドが得られる。このオペランドと減算器1227の出力とが、出力フォーマットシフタ1186と1187への制御信号を与える。

【0264】ブロック1229はEOI（画像終了）マーカ位置の検出に用いられる。EOIマーカ自身はストリップ1171において削除されるが、ストリップ1171で削除される以前にEOIマーカに先立つ位置に存在していたデータの最終ビットとなるパディングビットは存在する。比較器1229では、レジスタ1223に格納されているデータレジスタ1182中のビット数が8以下であるかどうかをチェックする。8以下であれば、ストリップ1171から新しいデータは入力されず（データレジスタ1182が復号されるデータ部の残りのビットを保持している）、残りのビットが削除されたEOIマーカの前のパディング領域サイズを示すことになる。さらなるパディング領域の処理やパディングビットの削除などは、上述のRSTマーカの前のパディングビットの場合に用いた手順と同一である。

【0265】パレルシフタ1186、1187と出力フォーマット部1188とはサポートする投割を有し、実施例に応じたさまざまな実装を考慮することができる。また、まったく実装されないこともあり得る。これらへの制御信号は上述のように制御部1185より与えられる。付加ビットプレシフタ1186はデータレジスタから32ビットを入力し、現在復号されているハフマン符号長だけ左にシフトする。このようにして、現在復号されている符号に続くすべての付加ビットは、パレルシフタ1186の出力に合わせて左に位置することになり、パレルシフタ1187への入力として送られる。付加ビットポストシフタ1187は、データの出力フォーマットとして用いられ図91にも示されている11ビットフィールドにおいて、左整列から右整列に付加ビット位置を調整する。付加ビットフィールドは出力ワードフォーマット1196においてビット8からビット18に拡張され、実際の付加ビット数に応じて最大ビットのいくつかは無効であることもある。このビット数はJPEG標準で規定されているように1196のビット0から3に符号化される。出力データフォーマットとして異なるフォーマットを用いる場合には、フォーマットに応じてパレルシフタ1186、1187とその機能を変更するこ

とになる。

【0266】出力フォーマットブロック1188は復号値をパックする処理を行い、JPEG標準では制御部1185から与えられるDC/AC係数（1196、ビット0から7）とDC係数指示ビット（1196、ビット19）、付加ビットポストシフタ1187から与えられる付加ビット（1196、ビット8から18）、マーカレジスタ1183から与えられるマーカ位置ビット（1196、ビット23）とを図91に示すフォーマットに従ってワードに構成する処理を行う。出力フォーマット部1188は、復号部の出力インタフェースに関する機能要件にも対処する。出力フォーマット部の実装は、異なる機能要件の結果として出力インタフェースを変更することになると、通常それに応じて変更される。上述のハフマン復号器は非常に効果的な復号処理を提供し、高速復号処理を実現する。

【0267】3. 17. 8 画像変換命令

これらの命令はソース画像の一般アフィン変換を行うためのものである。変換画像の一部を生成する処理は大きく2つのエリアに分けられる。一つはソース画像のどの部位が現在の出力スキャンラインと関連するかを決定するステップ、もう一つは必要なサブサンプリング/補間処理を行ってピクセルごとに出力画像を生成するステップである。

【0268】図92は、ソース画像の適切な領域が復号されているものとして、目的ピクセル値を計算するために必要なステップ720のフローチャートを示している。まず、サブサンプリングが行われていればサブサンプルが721で考慮される。次に、他の補間処理722と他のサブサンプリング処理といった2つの処理が通常実装されている。通常、補間とサブサンプリングとは別のステップであるが、補間とサブサンプリングとを一緒に行う場合もある。補間処理においては、まず周囲の4ピクセルを探し、ブレ乗算723が必要であるかどうかを、双線形補間724を行う前に決定する。双線形補間処理724は一般に計算量が非常に多くなるため、これにより画像変換処理動作が制約される。目的ピクセル値を計算する最後のステップは、ソース画像から双線形補間されたサブサンプルを加算する処理である。加算されたピクセル値はさまざまな方法で積分727され、目的画像ピクセル728が生成される。

【0269】画像変換命令のための命令ワード符号を図93に示すとともに、マイナーオPCODEフィールドの説明を以下の表に示す。

命令ワード：マイナーオPCODEフィールド

【0270】

【表19】

表 19: 命令ワード: マイナーオペコードフィールド

フィールド	説明
S	0=4周辺ソース画像ピクセルに対して双線形補間を行い実際のサンプル値を計算 1=サンプル値は最も近い画像ピクセル値
off[3:0]	0=オフセットレジスタ(mdp_por)の対応チャネルへの適用なし 1=オフセットレジスタ(mdp_por)の対応チャネルへの適用あり
P	0=ソース画像ピクセルのブレ乗算なし 1=ソース画像ピクセルのブレ乗算あり
C	0=出力値のクランプ処理なし 1=出力アンダーフローを 0x00、オーバフローを 0xFF にするクランプ処理あり
A	0=出力値の絶対値の考慮なし 1=ラップ処理あるいはクランプ処理の前に出力値の絶対値の考慮あり

【0271】 命令オペランドや結果フィールドの説明を以下に示す。

命令オペランドと結果ワード

【0272】

【表20】

表 20: 命令オペコードおよびリザルトワード

オペランド	説明	内部フォーマット	外部フォーマット
Operand A	kernel descriptor	-	短い或は長いカーネル記述子テーブル
Operand B	Source Image Pixels	other	画像テーブルフォーマット
Operand C	unused	-	-
Result	pixels	pixels	バックされたストリーム、バックされていないバイト

【0273】 オペランドAは、実際の変換を定義するために必要なすべての情報を記述している「カーネル記述子」として知られているデータストラクチャを指す。このデータストラクチャは2つのフォーマットのうちの1つとなる(A記述子のLビットで定義される)。図94はカーネル記述子の長い符号フォーマットを示し、図95は短い符号フォーマットを示す。カーネル記述子は、以下の情報を記述する。

1. ソース画像開始座標730(符号なしの固定長、24.24解像度)。位置(0,0)が画像の左上。
2. 水平731と垂直732(サブサンプル)デルタ(2の補数、固定長、24.24解像度)
3. 後述の固定長行列係数中のバイナリポイントの位置を示す3ビットのbpフィールド733
4. (存在する場合)は)積分行列係数735。これらは、bpフィールドによって暗黙的に指定されたバイナリ点の位置である20のバイナリ点の「可変」ポイント解像度(2の補数)である。
5. カーネル記述子中の残りのワード数を示すr1フィールド736。この値は列数と行数とを掛けたものから1を引いた値となる。

【0274】 記述子のカーネル係数は列ごとに並べられるが、ジグザグスキャンとなるように隣合う列は逆方向に並べられる。図96において、オペランドBはソース画像のスキャンラインを指すインデックステーブルへのポインターから成る。インデックステーブルの構造は図96に示されているように、オペランドB740がインデックステーブル741を指し、インデックステーブルが必要なソース画像ピクセルのスキャンライン(例えば742)を指すという構造である。一般に、インデックステーブルとソース画像ピクセルとはキャッシュ可能であり、ローカルメモリに位置している。

【0275】 オペランドCは水平/垂直サブサンプルレートを保持している。水平/垂直サブサンプルレートは、C記述子が存在する際に指定されるサブサンプル重み行列の次元によって定義される。行列rとcの次元は、図97に示すように画像変換命令のデータワードに符号化されている。結果ピクセルP[N]のチャネルNは以下の式に基づいて計算される。

【0276】

【数4】

$$p[n] = (l.offset[n] \cdot mdp_{por}:0000) + \sum_r \sum_c w_{r,c} \cdot s(x+r\Delta x, y+c\Delta y)[n]$$

【0277】内部的には、積分値は各チャネルごとの36のバイナリ点として保持される。フィールド中のバイナリ点の位置は、BPフィールドによって指定される。BPフィールドは削除する積分結果の先のビット数を示している。36ビットの積分値は符号付きの2の補数として表現され、指定されたようにクランプ処理あるいはラップ処理される。図98に、係数符号におけるBPフィールドの解釈例を示す。

【0278】3.17.9 畳込み命令
レンダリング画像に適用される畳込み処理は、2次元畳込みカーネルをソース画像に適用して結果画像を生成するものである。畳込み処理は通常、エッジ先鋭化やいろ

いろな画像フィルタにおいて用いられる。畳込み処理はコプロセッサ224において実装され、画像変換処理ではカーネルが各出力ピクセルごとにカーネル幅だけ移されるのに対し、畳込み処理では各出力ピクセルごとに1ソースピクセルが移動するといった点以外は、画像変換処理と同様の処理である。

【0279】ソース画像が値 $S(x, y)$ を有し、 $n \times m$ 畳込みカーネルが値 $C(x, y)$ を有すると、 S と C の畳込み $H[n]$ の n 番目のチャネルは、

【0280】
【数5】

$$H(x, y)[n] = (l.offset[n] \cdot mdp_{por}:0000) + \sum_i \sum_j S(x+i, y+j) \cdot C(i, j)[n]$$

【0281】で与えられる。ここで、 $i \in [0, c]$ 、 $j \in [0, r]$ である。オフセット値の意味、中間結果の解像度、bpフィールドの意味は画像変換命令と同一である。図99は、畳込みカーネル750がソース画像751に適用し、結果画像752を生成する例を示した図である。ソース画像アドレス生成や出力ピクセル計算は、画像変換命令と同様に行われる。命令オペランドも

画像変換と同様の形式である。図100は、畳込み命令の命令ワード符号を示したものであり、以下の表が種々のフィールドの説明である。

【0282】命令ワード
【0283】
【表21】

表 21: 命令ワード

フィールド	説明
S	0=4周辺ソース画像ピクセルに対して双線形補間を行い実際のサンプル値を計算 1=サンプル値は最も近い画像ピクセル値
C	0=出力ベクトル値のクランプ処理なし 1=出力ベクトルアンダーフローを 0x00、オーバーフローを 0xFF にするクランプ処理あり
P	0=ソース画像ピクセルのブレ乗算なし 1=ソース画像ピクセルのブレ乗算あり
A	0=出力値の絶対値の考慮なし 1=ラップ処理あるいはクランプ処理の前に出力値の絶対値の考慮あり
off[3:0]	0=オフセットレジスタの本チャネルへの適用なし 1=オフセットレジスタの本チャネルへの適用あり

【0284】3.17.10 行列乗算
行列乗算は、2つの色空間においてアフィン変換の関係が存在するような色空間変換処理などに用いられる。行列乗算は以下の式で定義される。

【0285】
【数6】

$$\begin{bmatrix} r_x \\ r_y \\ r_z \\ r_o \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ a_z \\ a_o \\ 1 \end{bmatrix}$$

【0286】行列乗算命令オペランドと結果ワードは以下のフォーマットを有する。
命令オペランドと結果ワード

【0287】
【表22】

表 22: 命令オペコードおよびリザルトワード

オペランド	説明	内部フォーマット	外部フォーマット
Operand A	ソース画像画素	画素	バックされたストリーム
Operand B	行列 係数	その他	画像テーブルフォーマット
Operand C	未使用	-	-
Result	画素	画素	バックされたストリーム、バックされていないバイト

【0288】図 10-1 に行列乗算命令のための命令ワード
 符号を示すとともに、以下の表にマイナーオペコード
 フィールドを示す

【0289】

【表 23】

表 23: 命令ワード

フィールド	説明
C	0=出力ベクトル値のクランプ処理なし 1=出力ベクトルアンダーフローを 0x00、オーバーフローを 0xFF にするクランプ処理あり
P	0=画像ピクセルのブレ乗算なし 1=画像ピクセルのブレ乗算あり
A	0=出力値の絶対値の考慮なし 1=ラップ処理あるいはクランプ処理の前に出力値の絶対値の考慮あり

【0290】3. 17. 11 ハーフトーン化
 コプロセッサ 224 はハーフトーン処理のための多値レベルディザを備える。2 から 255 までの値は意味のあるハーフトーンレベルとなる。ハーフトーンするデータは、スクリーンが対応してメッシュあるいはアンメッシュである限り、バイト（アンメッシュあるいはメッシュデータからの 1 チャネル）あるいはピクセル（メッシュ）のどちらでも良い。4 つの出力チャネル（あるいは同一チャネルから 4 バイト）まで、一緒にバックされたようなあるいはバイトごとに 1 符号にアンパックされたようなバックビット（2 レベルハーフトーンの場合）あ

るいは符号（2 出力レベル以上の場合）生成することができる。

【0291】出力ハーフトーン値は以下の式を用いて計算される。

$$(P \times (1 - 1) + d) / 255$$

ここで、p はピクセル値 ($0 \leq p \leq 255$)、l はレベル数 ($2 \leq l \leq 255$)、d はディザ行列値 ($0 \leq d \leq 254$) である。オペランド符号は以下の通りである。

命令オペランドと結果ワード

【0292】

【表 24】

表 24: 命令オペコードおよびリザルトワード

オペランド	説明	内部フォーマット	外部フォーマット
Operand A	ソース画像の画素	画素	バックされたストリーム
	ソース画像バイト	バックされたバイト、バックされていないバイト	バックされたストリーム
Operand B	dither matrix coefficients	画素、バックされたバイト、バックされていないバイト	バックされたストリーム、バックされていないバイト
Operand C	未使用	-	-
Result	ハーフトーンコード	画素、バックされたバイト、バックされていないバイト	バックされたストリーム、バックされていないバイト

【0293】命令ワード符号では、マイナーオペコードはハーフトーンレベル数を指定する。オペランド B 符号

はハーフトーンスクリーンのためのものであり、タイル合成と同様に符号化される。

3. 17. 12 階層的画像フォーマット復号

階層的画像フォーマット復号処理は複数のステップを含む。これらのステップは、水平補間、垂直補間、ハフマン復号、残部融合である。各ステップは別の命令でもって実行される。ハフマン復号ステップでは、補間ステップからの補間された値に付加される残りの値がハフマン符号化される。従って、J P E G復号部がハフマン復号において用いられる。

【0294】図102に、水平補間処理を示す。出力ストリーム761は入力ストリーム672の2倍のデータとなり、最後のデータ値763は複製されている764。図103は4倍の水平補間を行う例である。階層的画像フォーマット復号の第2ステップでは、線形補間によりピクセル列を2倍あるいは4倍に垂直にアップサンプルする。このステップでは、1ピクセル列がオペランドA、他の列がオペランドBとなる。

【0295】垂直補間の場合には2倍、4倍どちらの場合でも、出力データストリームは入力ストリームと同数のピクセルとなる。図104に、2つの入力データスト

リーム770、771を用いて2倍補間の出力ストリーム772と4倍補間の出力ストリーム773を生成する垂直補間の例が示されている。ピクセル補間の場合には、補間処理は4つのチャネルピクセルの4チャネルごとに別々に行われる。

【0296】残部融合処理は、2つのデータストリームのバイトごとの加算を含む。第一ストリーム（オペランドA）はベース値ストリームであり、第二ストリーム（オペランドB）は残値ストリームである。図105に、残部融合処理を用いた場合の2つの入力ストリーム780、781と対応する出力ストリーム782を示す。

【0297】図106は、階層的画像フォーマット命令の命令ワード符号を示したものであり、以下の表にマイナーオペコードフィールドの詳細を示す。

命令ワード—マイナーオペコードフィールド

【0298】

【表25】

表 25: 命令ワード:マイナーオペコードフィールド

フィールド	説明
R	0 = 補間 1 = 残りのマージ
V	0 = 水平補間 1 = 垂直補間
F	0 = 2 のファクタの補間 1 = 4 のファクタの補間
C	0 = 結果の値をクランプしない 1 = 結果の値をクランプする: underflow to 0x00, overflow to 0xFF

【0299】3. 17. 13 命令コピー命令
これらの命令は2つのそれぞれ別のグループに分けられる。

a. 汎用データ移動命令

これらの命令は、入力インタフェースモジュール、入力インタフェーススイッチ252、ピクセルオーガナイザ246、J P E G符号化部241、結果オーガナイザ249、出力インタフェースモジュールからなるコプロセッサ224内の通常のデータフローパスを用いる。この場合、J P E G符号化モジュールはデータを処理を行わずに直接送る。

【0300】データ操作動作の他の命令としては以下のものが挙げられる。

- ・サブバイト値（ビット、2ビット値、4ビット値）のバイトへのパッキング、アンパッキング
- ・ワード内でのバイトのパッキングとアンパッキング
- ・整列
- ・バイトレインスワッピングと複製
- ・メモリクリア
- ・値の複製

データ操作動作は、ピクセルオーガナイザ（入力）と結果オーガナイザ（出力）の組み合わせで実行される。多くの場合、これらの命令は他の命令と組み合わせて用いられる。

b. ローカルDMA命令

データ操作は行われない。図2に示すように、ローカルメモリ236と周辺インタフェース237間でデータ転送（双方向）が行われる。これらの命令は実行が他の命令とオーバーラップする唯一の命令である。最大これらの命令の1つが「オーバーラップしていない」命令と同時に実行することができる。

【0301】メモリコピー動作では、オペランドAはコピーするデータを示し、結果オペランドはメモリコピー命令の目的アドレスを示す。汎用のメモリコピー命令では、オペランドBによって入力へのデータ操作動作が規定され、オペランドCによって出力オペランドワードへの動作が規定される。

3. 17. 14 フロー制御命令

フロー制御命令は、図9に示したような命令実行モデルのさまざまな部位を制御するための命令群である。フロ

一制御命令としては、命令ストリームを実行しちえるときに1つの仮想アドレスから他のアドレスへの移動を可能にする条件付きジャンプあるいは条件なしジャンプを含む。条件付きジャンプ命令は、コプロセッサやレジスタでもって関連するフィールドをマスクし、所定の値と比較することにより決定される。これにより命令の一般性を保つことができる。更に、フロー制御命令は、オーバラップ命令と非オーバラップ命令との間の同期をとる

ために、あるいはマイクロプログラミングの一部として用いられる待機命令をも含む。

【0302】図107に、フロー制御命令の符号を示す。また、以下の表はマイナーオペコードの説明である。

命令ワードマイナーオペコードフィールド

【0303】

【表26】

表 26: 命令ワード:マイナーオペコードフィールド

フィールド	説明
type	00 = ジャンプ 01 = ウェイト
C	0 = 無条件ジャンプ 1 = 条件ジャンプ
S	0 = オペランド B を条件レジスタとして使用し、オペランド C を条件マスクとして使用 1 = 如何なる割込みの条件をセット
N	0 = 条件が真の時にジャンプ 1 = 条件が真でない時にジャンプ
O	0 = 重複していない命令が終了するのを待つ 1 = 重複した命令が終了するのを待つ

【0304】ジャンプ命令においては、オペランドAワードはジャンプ命令の目的アドレスを指定する。マイナーオペコードのSビットが0にセットされれば、オペランドBはコプロセッサレジスタを指定し、条件のソースとして用いる。オペランドB記述子の値はレジスタのアドレスを指定し、オペランドBワードの値がレジスタ内容を比較する値となる。オペランドCワードは結果に適用されるビットごとのマスクを指定する。すなわち、ジャンプ命令条件は以下のビットごとの式が満たされていれば真となる。

【0305】 $((\text{register_value} \text{ x or Operand B}) \text{ and Operand C}) = 0 \times 00000000$

更に、マイクロプログラミングレベルで十分に制御するためのレジスタアクセスのためにも当該命令が用いられる。

3. 18 アクセラレータカードのモジュール

図2において、種々のモジュールを更に説明する。

【0306】3. 18. 1 ピクセルオーガナイザ
ピクセルオーガナイザ246は入力インタフェーススイッチ252からのデータストリームのアドレスを指定してバッファに格納する。入力データはピクセルオーガナイザの内部メモリに格納されるか、あるいはMUVバッファ250に格納される。入力ストリームに対する必要なデータ処理を全部済ませた後、必要に応じて入力ストリームを主データバス242あるいはJPEG符号化器241に渡す。ピクセルオーガナイザの動作モードは通常のCBUSインタフェースによって構成することができる。ピクセルオーガナイザ246はPO_CFG制御レジスタの指定するような五つのモードのうちの一つ

のモードで動作する。これらのモードは次のとおりである。

(a) アイドルモード：ピクセルオーガナイザ246が動作しないモード。

(b) シーケンシャルモード：入力データは内部FIFOに格納されるようになり、ピクセルオーガナイザ246はデータの32ビットアドレスを生成して入力インタフェーススイッチ252にデータを要求するモード。

(c) 色空間変換モード：ピクセルオーガナイザが色空間変換のためにピクセルをバッファするモード。更に、MUVバッファ250に格納されているインターバルおよび分数値を要求する。

(d) JPEG圧縮モード：ピクセルオーガナイザ246が画像データをMCUの形式でMUVバッファに格納するモード。

(e) 畳込み演算および画像変換モード：ピクセルオーガナイザ246が行列係数をMUVバッファ250に格納し、必要であれば主データバス242にもそれを伝えるモード。

【0307】ピクセルオーガナイザ246は主データバス242とJPEG符号化器241の両方とも動作のためにMUVバッファ250を使う。色空間変換において、インターバルおよび分数テーブルはMUV RAM 250によって格納され、36ビットのデータ(4つのカラーチャネル)×(4ビットのインターバル値と8ビットの分数値)としてアクセスされる。画像変換および畳込み演算のために、MUV RAM 250は行列係数および関連する構成データを格納する。係数行列は16行×16列に制限され、各係数の幅は最大20ビットである。MUV RAM 250は1クロックサイクルあたり

1つの係数を必要とする。係数データに加えて、バイナリポイント、ソーススタート座標、サブサンプルデルタ等の制御情報も主データバス242に伝えなければならない。この制御情報は、行列係数より先にピクセルオーガナイザ246によってフェッチされる。

【0308】JPEG圧縮において、ピクセルオーガナイザ246は、MUVバッファ250を使ってMCUをダブルバッファする。JPEG圧縮の性能向上のためには、ダブルバッファ技術を使うことが望ましい。MUV

RAM250の1半分は入力インタフェーススイッチ252からのデータを使って書き込まれる。一方、もう一方の半分は、JPEG符号化器241に送るべきデータを得るためにピクセルオーガナイザによって読み出される。ピクセルオーガナイザ246は、必要とされる所におけるカラー成分の水平サブサンプリングを行うとともに、入力画像のサイズがMCUの整数倍でない場合にはMCUをパディングする。

【0309】ピクセルオーガナイザ246は、図32において前述した、バイトレインスワップと、正規化と、バイト入り代えと、バイトパックおよびアンパックと、複写動作とを含む入力データのフォーマットをも行う。動作はピクセルオーガナイザレジスタを設定することにより必要に応じて行われる。図108において、ピクセルオーガナイザ246をより詳細に説明する。ピクセルオーガナイザ246は、CBUSインタフェース制御部801に含まれている自身のレジスタセットの制御に従い作動しており、CBUSインタフェース制御部801はグローバルCBUSを経由して命令制御部235に接続されている。ピクセルオーガナイザ246にはオペランドフェッチ部802が含まれており、ピクセルオーガナイザ246が必要とするオペランドデータを入力インタフェーススイッチ252から要求する。、オペランドデータのスタートアドレスは、実行直前にセットされるPO_SAIDレジスタによって指定される。PO_SAIDレジスタは、PO_DMRレジスタのLビットによる指定に応じて、即座のデータを保持することもある。現在アドレスポインタはPO_CDPレジスタに格納され、入力インタフェーススイッチの要求があればそのバースト長さだけ増加される。データがMUV RAM250にフェッチされるとき、データの現在オフセットはPL_MUVレジスタによって指定されるMUV RAM250のベースアドレスと連結される。

【0310】オペランドフェッチ部802によってフェッチされたシーケンシャル入力データをバッファするために、FIFO803が用いられる。データ操作部804は、図32において説明したような様々な操作を実行する。データ操作部の出力はMUVアドレス生成部805に伝えられる。MUVアドレス生成部805は構成レジスタに従ってデータをMUV RAM250、主データバス242、JPEG符号化器241のどちらかに伝

える。ピクセルオーガナイザ制御部806は、ピクセルオーガナイザ246のサブモジュール全てのために必要な制御信号を生成する状態機械である。必要な信号の中では、種々のBUSインタフェース上での通信を制御する信号も含まれる。ピクセルオーガナイザ制御部は、状態レジスタの設定に従い他モジュール239が必要とする診断情報を出力する。

【0311】図109において、図108のオペランドフェッチ部802をより詳細に示す。オペランドフェッチ部802には、命令バスアドレス生成部(IAG)810が含まれており、オペランドデータをフェッチせよという要求を生成する状態機械を含む。この要求は要求仲裁部811に送られ、要求仲裁部811はアドレス生成部810の要求とMUVアドレス生成部805の要求(図108)との間を仲裁しており、勝ちの要求を入力(MAG)インタフェーススイッチ252に送るようにしている。要求仲裁部811は要求を扱うための状態機械を含んでいる。これは、FIFOカウンタ部814を用いてFIFOの状態をモニタし、次の要求をいつデスパッチすべきかを決定する。バイトイネーブル生成部812はIAG810の情報を受け取り、入力インタフェーススイッチ252がリターンする各オペランドにおける有効なバイトを指定するバイトイネーブルパタン816を生成する。バイトイネーブルパタンは関連するオペランドデータとともにFIFOに格納される。MAG要求とIAG要求が同時に到着したとき、要求仲裁部811はMAG要求をIAG要求より優先して処理する。

【0312】図108において、MUVアドレス生成部805は異なるいくつかのモードで動作する。これらのモードにおいて、第1はJPEG(圧縮)モードである。このモードでは、JPEG圧縮のための入力データがデータ操作部804によって供給され、MUVバッファ250はダブルバッファとして使われる。MUV RAM250アドレス生成部805は、データ操作部804によって処理された入力データを格納するに適するMUVバッファのアドレスを生成する。MAG805は、格納されたピクセルからカラー成分データを取り出すための読み出しアドレスを生成するとともに、JPEG圧縮用の8×8ブロックを形成するように動作する。MAG805は、MCUが画像と一部重なっている場合も扱う。図110は、MAG805が行うパディング動作の一例を示す。

【0313】普通のピクセルデータにおいて、MAG805は、4つの8ビットRAMのMUV RAM250における同じアドレス内に、4つのカラー成分を格納する。同じカラーチャネルからデータを同時に取り出すために、MCUデータは左にバレルシフトされてからMUV RAM250に格納される。データの左にシフトされるバイト数は、書き込みアドレスの下位2ビットによって決定される。例えば、図111は、サブサンプリン

グの要らない場合32ビットピクセルデータがMUV RAM250内で配置されるデータ構造を示す。3チャンネル又は4チャンネルインターリーブJPEGモードにおいては、入力データのサブサンプリングが選択されることもあり得る。サブサンプリングを伴うマルチチャンネルJPEG圧縮モードにおいて、MAG805(図108)は、JPEG符号化器の最適性能のために32ビットデータがMUV RAM250に格納される前にサブサンプリングを行うようになっている。最初四つの入力ピクセルの中で、MUV RAM250に格納される第1および第4番目のチャンネルだけが有用なデータを含んでいる。第2および第3番目のチャンネルのデータはサブサンプリングされ、ピクセルオーガナイザ246のレジスタに格納される。次の4つの入力ピクセルにおいて、第2および第3番目のチャンネルはサブサンプリングされたデータをもって埋められる。図112は、マルチチャンネルサブサンプリングモードにおけるMCUデータ構成の一例を示す。MAGは単一チャンネルアンパックデータ全てをマルチチャンネルピクセルデータと全く同様に扱う。MUV RAMから読み出された単一チャンネルパックデータの一例が図113に示されている。

【0314】書き込みプロセスによって入力MCUがMUV RAMに格納されている間、読み出しプロセスはMUV RAMから8×8ブロックを読み出す。一般的に、前記ブロックは各チャンネルに対してデータを順次読み出すことによって、四つの係数ずつMAG805によって生成される。ピクセルデータとアンパック入力データにおいて、格納されるデータは図111に示すように整理される。従って、サンプルされなかったピクセルデータからなる8×8ブロックを合成するためには、読み出しプロセスはMUV RAMからデータを斜行しながら読み出す。図114は、このようなプロセスの一例を示す。図114には、四つのチャンネルデータにおける読み出しシケンス示されており、MUV RAM250の格納形式が同一チャンネルから多数の値を同時に読み出すことを容易にしていることが分かる。

【0315】色変換モードにおいて、MUV RAM250はインターバルおよび分数値を格納するキャッシュとして用いられ、MAG805はそのキャッシュの制御部として働くようになっている。MUV RAM250は3つのカラーチャンネル値をキャッシュする。ここで、各カラーチャンネルは256対の4ビットインターバルおよび分数値を有する。DMUを通じた各ピクセル出力において、MUV RAM250から前記値を得るためにMAG805が使われる。この値が得られないときに、MAG805は欠けているインターバルおよび分数値をフェッチせよというメモリ読み出し要求を出す。帯域の有効利用のために、要求あたりエントリー一つだけをフェッチする手法のかわりに、多数のエントリーをフェッチするような手法を取る。

【0316】画像変換および畳込み演算のために、MUV RAM250はMDPの行列係数を記憶している。MAGはMUV RAM250に格納されている全ての行列係数をスキャンする。画像変換および畳込み命令の始めにおたつて、MAG805はオペランドフェッチ部に要求を出し、オペランドフェッチ部がカーネル記述“ヘッダ”(図94)とバスト要求の第1行列係数とをフェッチするようにする。

【0317】図115において、図108のMUVアドレス生成部(MAG)805をより詳細に示す。MAG805はIBus要求を多重化するIBus要求モジュール820を備えており、IBus要求は画像変換制御部(ITX)821と色空間変換(CSC)制御部822によって生成される。この要求は、要求を実行するようになっているオペランドフェッチ部に送られる。ピクセルオーガナイザ246は画像変換、色空間変換のどちらか1つのモードで動作するようになっているため、制御部821、822の間では仲裁が要らないことになる。IBus要求モジュール820は、オペランドフェッチ部への要求を生成するのに必要なバストアドレスとバスト長さを含む情報を、関連するピクセルオーガナイザから導出する。

【0318】JPEG制御部824は、JPEG書き込み制御部とJPEG読み出し制御部という2つの状態機械を備えており、JPEGモードにおいて使われる。前記二つの制御部は同時に作動するようになっており、内部レジスタを用いることによってお互いに同期を取る。JPEG圧縮動作において、DMUはMCUデータを出しMUV RAMに格納する。JPEG書き込み制御部は水平パディングとピクセルサブサンプリングの制御とを担当しており、JPEG読み出し制御部は垂直パディングを担当する。水平パディングはDMU出力を停止することによって行われ、垂直パディングは既に読み出した8×8ブロックを再び読み出すことによって行われる。

【0319】JPEG書き込み制御部は、ソース画像におけるDCUおよびDMU出力ピクセルの現在位置をトラッキングしており、水平パディングのためにいつDMUを停止すべきかを決定するのにその情報を用いる。MCUがMUV RAM250に書き込まれたときに、JPEG書き込み制御部は内部レジスタをセットするかまたはリセットすることによって、MCUが画像の右エッジにあるかあるいは画像の最低エッジにあるかを表す。JPEG読み出し制御部は、前記レジスタの内容に基づき、垂直パディングが必要であるかや画像の最後のMCUまで読んだのかを判断する。

【0320】JPEG書き込み制御部はDMU出力データをトラッキングし、DMU出力データをMUV RAM250に格納する。前記制御部は、レジスタセットを用いて入力ピクセルの現在位置を記憶する。この情報は

DMU出力を停止して水平バディंगを行うときに使われる。全てのMCUがMUV RAM250に書き込まれたときに、前記制御部はMCU情報をJPEG-RW-IPCレジスタに書き込み、以後JPEG読み出し制御部によって利用し得るようにする。

【0321】この制御部は、最後のMCUがMUV RAM250に書き込まれた後、SLEEP状態に入り現在の命令が終了するまでその状態に残る。JPEG読み出し制御部は、MUV RAM250に格納されているMCUから8×8ブロックを読み出す。マルチチャンネルピクセルにおいては、制御部がMCUを数回に渡って読み出すようになっており、MUV RAMに格納されている各ピクセルから、各読み出しにおける異なるバイトを抽出する。

【0322】この制御部はJPEG-RW-IPCによって提供される情報を用いて、垂直バディंगを行うべきかを検出する。垂直バディंगはMUV RAM250から読み出した直前の8バイトを再び読み出すことによって行われる。画像変換制御部821はIBusからカーネルディスクリプタを読み出し、カーネルヘッダをMDP242に伝える。そして、po.lenレジスタで指定された回数だけ行列係数をスキャンする。画像変換および畳込み命令において、PO246による全てのデータ出力はIBusから直接フェッチされるようになっており、DMUには伝えられない。

【0323】カーネルヘッダの直後フェッチされる第1行列係数の最初8ビットは、フェッチすべき残りの行列係数の数を表す。カーネルヘッダは修正されずに直接MDPに伝えられるが、行列係数はMDPに伝えられる前にサイン拡張される。ピクセルサブサンプラ825は、それぞれが入力ワードの1バイトに対して動作する二つの同じチャンネルサブサンプラを備える。関連する構成レジスタが起動されていないときに、ピクセルサブサンプラは自身の入力をそのまま自身の出力にコピーする。一方、構成レジスタが起動されているときに、サブサンプラは入力データに対して平均を取るか又は間引きを行うかすることによって入力データをサブサンプルする。

【0324】MUV多重化モジュール826は現在アクティブである制御部からMUV読み出しおよび書き込み信号を選ぶ。内部多重化部は、MUV RAM250を使う種々の制御部を経由して、読み出しアドレス出力を選ぶ。MUV RAM書き込みアドレスはMUV多重化モジュールの8ビットレジスタに格納されている。MUV RAM250を用いる制御部は次のMUV RAMアドレスを決定するための制御を行うとともに、書き込みアドレスレジスタをロードする。

【0325】MUV有効アクセスモジュール827は色空間変換制御部によって用いられ、データ操作部による現在ピクセル出力のインターバルおよび分数値がMUV RAM250において利用できるかを決定する。一つ

以上のカラーチャンネルが欠けているとき、MUV有効アクセスモジュール827は関連するアドレスをIBus要求モジュール820に伝え、インターバルおよび分数値をバーストモードでロードする。キャッシュミスがサービスされると、MUV有効アクセスモジュール827は今までフェッチされたインターバルおよび分数値のセットを表す内部有効ビットをセットする。

【0326】複写モジュール829は、内部ピクセルレジスタが定める回数だけ、入力データを複写する。複写モジュールが現在の入力ワードを複写している間、入力ストリームは停止されるようになる。PBUSインタフェースモジュール830は、ピクセルオーガナイザ246を主データバス242およびJPEG符号化器241にリタイムするか或いはその逆の処理をするのに使われる。最後に、MAG制御部831は種々のサブモジュールをイニシエイトする信号とシャットダウンする信号とを生成する。なお、MAG制御部831は、主データバス242およびJPEG符号化器241からの入力PBUS信号に対する多重化をも行う。

【0327】3.18.2 MUVバッファ

図2においては、これまでの説明から明らかなようにピクセルオーガナイザ246はMUVバッファ250と相互関係にある。再コンフィギュレーション可能なMUVバッファ250は単純ルックアップテーブルモード（モード0）、多重ルックアップテーブルモード（モード1）、JPEGモード（モード2）を含む様々な処理モードをサポートしている。それぞれのモードで、バッファには異なるタイプのデータオブジェクトが格納される。例えば、バッファに格納されているデータワード、様々な検索テーブルの値、単一チャンネルデータ、複数チャンネルデータはデータオブジェクトである。一般的に、データオブジェクトは異なるサイズを持つ。更に再コンフィギュレーション可能なMUVバッファ250に格納されたデータオブジェクトはバッファのオペレーティングモードに依存した様々な方法で実際にアクセスできる。

【0328】異なるタイプのデータを書き戻したり及び格納するのに必要な様々な方法を適切にするために、データオブジェクトはしばしば、格納される前に符号化される。データオブジェクトのコーディングに用いられる方法はデータオブジェクトのサイズ、表現されているデータオブジェクトのフォーマット、どのようにデータオブジェクトがバッファから書き戻されるのか、バッファ上に形成されたメモリモジュールの構成状態によって決定される。

【0329】図116は再コンフィギュレーション可能なMUVバッファ250を実装するために用いられるコンポーネントのブロックダイアグラムである。再コンフィギュレーション可能なMUVバッファ250はエンコーダ1290、ストレージデバイス1293、デコーダ

1291、アドレス読み込み・ローテーション信号発生器1292からなる。入力データストリーム1295にデータオブジェクトが入力された時には、データオブジェクトはエンコーダ1290により内部データに符号化され、内部データストリーム1296に配置される。符号化されたデータオブジェクトはストレージデバイス1293に格納される。

【0330】格納されたデータオブジェクトを復号化する場合には、符号化されたデータは符号化データ出力ストリーム1297によりストレージデバイスから取り出される。符号化データ出力ストリーム1297上の符号化されたデータはデコーダ1291によって復号化される。復号化されたデータオブジェクトは出力データストリーム1298上に現れる。

【0331】ストレージデバイス1293への書き込みアドレス1035はMAG805 (図108) により与えられる。書き込みアドレス1299、1300、1301も同様にMAG805 (図108) によって与えられ、アドレス読み込み・ローテーション信号発生器1292によってストレージデバイス1293に分配される。アドレス読み込み・ローテーション信号発生器1292はまた、入力・出力ローテーション信号1303、1304をエンコーダ、デコーダそれぞれに対して生成する。書き込み有効信号1306と1307は外部ソースから与えられる。コントローラ801 (図108) によって与えられる処理モード信号1302はエンコーダ1290、デコーダ1291、アドレス読み込み・ローテーション信号発生器1292、ストレージデバイス1293に接続される。インクリメント信号1308はアドレス読み込み・ローテーション信号発生器内の内部カウンタをインクリメントし、JPEGモード (モード2) でも用いられることがある。

【0332】再コンフィギュレーション可能なMUVバッファ250が単純ルックアップテーブルモード (モード0) である場合には、本質的にバッファ250はむしろ、単一モードのメモリモジュールの様に動作する。データオブジェクトは本質的にメモリモジュールにアクセスする方法と同様な方法でバッファに格納あるいはバッファから取り出せる。

【0333】再コンフィギュレーション可能なMUVバッファ250が多重ルックアップテーブルモード (モード1) で動作中の時、バッファ250はストレージデバイス1293に格納されている最大3つの検索テーブルをもちいて複数のテーブルに分割される。検索テーブルは同時かつ独立にアクセスすることができる。一例を挙げると、インターバルおよびフラクションの値は多重ルックアップテーブルモードのストレージデバイス1293に格納される、テーブルは入力データストリーム1295の下位3バイトを利用してインデックスがつけられる。3バイトのそれぞれはストレージデバイス1293

に格納された独立の検索テーブルに発行される。

【0334】画像がJPEG圧縮されているとき、画像は符号化されたデータストリームに変換される。ピクセルは原画像からMCUのフォーマットで取り出される。MCUは画像の左から右に、上から下に読み出される。それぞれのMCUは多数の8×8のブロックに再合成される。多数の8×8ブロックはMCUから抽出される。MCUは原画像のカラーコンポーネント、複数チャネルのJPEGモード、サブサンプリングの必要性等のいくつかの要因に依存している。8×8のブロックはその後フォワードDCT (FDCT)、量子化、エントロピー符号化される。JPEG圧縮の場合には、符号化されたデータはデータストリームからシーケンシャルに読み込まれる。データストリームはエントロピー復号化、逆量子化、逆DCT (IDCT) が行われる。IDCT処理の出力は8×8のブロックである。多数の8×8ブロックはMCUを再構成するように統合される。JPEG圧縮を用いるとき、多数の8×8ブロックは前述の要因に依存する。再コンフィギュレーション可能なMUVバッファ250はMCUを多数の8×8ブロックに分解したり、多数の8×8ブロックをMCUに再構成したりするときにも用いられる。

【0335】再コンフィギュレーション可能なMUVバッファ250がJPEGモードの処理を行っているときはバッファ250への入力データストリーム1295はJPEG圧縮処理を行っているピクセルあるいはJPEG圧縮処理を行っている単一のコンポーネントを含んでいる。バッファ250の出力データストリームはJPEG伸長処理の単一チャネルデータブロックあるいはJPEG伸長処理のピクセルデータを含んでいる。このJPEG圧縮の例では、入力ピクセルはY、U、V、Oの4チャネルまで構成できる。指定の数のピクセルが完成したピクセルブロックとして処理処理されたときには、単一のコンポーネントデータブロックの抽出が開始できる。それぞれの単一のコンポーネントデータブロックはバッファに格納された同チャネルのピクセルからなるデータにより構成される。従ってこの例では、4つまでの単一のコンポーネントデータブロックをひとつのピクセルデータブロックから抽出できる。この具体例では、再コンフィギュレーション可能なMUVバッファ250がJPEG圧縮用のJPEGモード (モード2) で処理を行っているときには、多数の単位最小コード (MCU) はそれぞれ64の単一あるいは複数チャネルのピクセルをバッファに格納でき、多数の64バイト長の単一チャネルのコンポーネントデータブロックをバッファに格納されたそれぞれのMCUから抽出できる。例えば、バッファ1289がJPEG伸長を行うためにJPEGモード (モード2) である間は、出力データストリームは、Y、U、V、Oの最大4つのコンポーネントを持つ出力ピクセルから構成される。要求された数の完成した単一

のコンポーネントデータブロックをバッファに書き込んだときは、ピクセルデータの抽出ができる。異なる色のコンポーネントに対応する4つの単一のコンポーネントデータブロックからのバイトは出力ピクセルとして取り出される。

【0336】図117は図116のエンコーダ1290の詳細図である。ピクセルブロックの伸長では、入力データオブジェクトそれぞれはストレージデバイス1293に格納される前にバイト方向のローテーションにより符号化される(図129)。ローテーションの大きさは入力ローテーション制御信号1303により決定される。この例ではピクセルデータが最大の4バイトであったときは、32ビットの4入力1出力のマルチプレクサ1320および1325が、4つのうちの1つの可能な入力ピクセルのローテーションの選択に用いられる。例えば、もしピクセルの4つのバイトが(3, 2, 1, 0)のようにラベルが付けられていたとすると、このピクセルのローテーションは(3, 2, 1, 0)(0, 3, 2, 1)(1, 0, 3, 2)(2, 1, 0, 3)となる。4つの符号化されたバイトはストレージデバイスの1290に出力される。

【0337】バッファがJPEGモード(モード2)以外のモード、例えば、単一ルックアップテーブルモード(モード0)、多重ルックアップテーブルモードである時には、バイト方向のローテーションは必要ではなく、また入力データオブジェクトに対して行えない。入力データオブジェクトは後者の場合に、ノーローテーションの値をもつ入力ローテーション制御信号を無視することによって、ローテーションにより妨害を受ける。この値1323はである。2入力1出力のマルチプレクサ1321は制御信号1326を入力ローテーション制御信号1303とノーオペレーション値1323の選択をすることによって生成する。現在の処理モード1302はマルチプレクサ選択信号を生成するために、ピクセルブロック分解モードの値と比較される。信号1326によって制御される4入力1出力のマルチプレクサ1320は入力データオブジェクトの4つのローテーションのうち1つを選択し、符号化された入力データストリーム1326上に符号化された入力データオブジェクトを生成する。

【0338】図118は符号化された出力データストリーム1297を復号化するデコーダ1291を実装する組み合わせ回路の回路図である。デコーダ1321はエンコーダと本質的に同様な方法で動作する。デコーダはデータバッファがJPEGモード(モード2)である場合にのみデータを操作する。下部の符号化されたデータストリーム1297内の符号化された出力データオブジェクトの下位32ビットはデコーダに渡される。データはエンコーダ1290でローテーションするのは逆の感覚でバイト方向のローテーションを用いて復号化され

る。32ビットの4入力1出力のマルチプレクサは、可能な4つの種類の符号化データのうちの1つを選択するために用いられる。例えば4バイトの入力ピクセルが(3, 2, 1, 0)のようにラベルが付けられているとすると、このピクセルのローテーションの種類は(3, 2, 1, 0)(2, 1, 0, 3)(1, 0, 3, 2)(0, 3, 2, 1)の4つが可能である。出力ローテーション制御信号1304はバッファがピクセルブロック分解ノードの時と、他のオペレーションモードでノーオペレーション値が無視されたときに使用される。ノーオペレーション値1333は0である。2入力1出力のマルチプレクサ1331は、出力ローテーション制御信号1304とノーオペレーション値1333の選択を行うことで信号1334を生成する。現在の処理モード1302はマルチプレクサ選択信号1332を生成するために、ピクセルブロック分解モードの値と比較される。信号1334②によって制御される4入力1出力のマルチプレクサ1330は符号化された出力データストリーム1297上の符号化された出力データオブジェクトの4種類のローテーションを選択し、出力データストリーム1298上に出力データを生成する。

【0339】図116において、回路で用いられる内部読み込みアドレス生成の方法は、再コンフィギュレーション可能なMUVバッファ250の処理モード1302によって選択される。単一ルックアップテーブルモード(モード0)と多重ルックアップテーブルモード(モード1)では読み込みアドレスは外部読み込みアドレス1299, 1300, 1301の形でMAG805(図108)によって生成される。単一ルックアップテーブルモード(モード0)ではストレージデバイス1293上にメモリモジュール1380, 1381, 1382, 1383, 1384, 1385(図121)は一緒に処理する。メモリモジュール1380から1385(図121)に与えられる書き込みアドレスと読み込みアドレスは本質的に同じである。即ち、ストレージデバイス1293は外部回路に1つの読み込みアドレスと1つの書き込みアドレスの供給のみを必要とし、これらのアドレスをメモリモジュール1380から空1385(図121)に分配するために内部ロジックを使用する。モード0では、読み込みアドレスは外部アドレス1299(図116)により与えられ、本質的に変化しないまま内部アドレス1348(図121)に分配される。外部読み込みアドレス1349, 1350, 1351(図121)はモード0では使用されない。書き込みアドレスは外部書き込みアドレス1305(図116)により与えられ、本質的に修正なしで各メモリモジュール1380から1385(図121)の書き込みアドレスに接続される。

【0340】ここでは、多重ルックアップテーブルモード(モード1)における3ルックアップテーブルの構成

を示す。3つのテーブルが独立にアクセスされるとき、符号化された入力データは1380から1385（図121）までのすべてのメモリもジュールに同時に書き込まれ、従って3つのテーブルそれぞれに1つのインデックスが必要となる。メモリモジュール1380から1385（図212）への3つのインデックス、即ち読み込みアドレスはストレージデバイス1293により与えられる。これらの読み込みアドレスは、内部ロジックを用いて1380から1385の適切なメモリモジュールに分配される。本質的に単一ルックアップテーブルモードのときと同様な手法で、外部から与えられる書き込みアドレスは、本質的な変更なしに1308から1385のそれぞれのメモリモジュールのアドレスに接続される。その結果、多重ルックアップテーブルモード（モード1）では外部読み込みアドレス1299、1300、1311は内部読み込みアドレス1348、1349、1350にそれぞれ分配される。内部読み込みアドレス1352はモード1では使用されない。JPEGモード（モード2）で使用される内部アドレス生成方法は前述の方法とは異なる。

【0341】図119はJPEG圧縮を行うJPEGモード（モード2）における、再コンフィギュレーション可能なデータバッファ用の、読み込みアドレスおよびローテーション信号生成回路1292を実装する組み合わせ回路の回路図である。JPEGモード（モード2）では、信号生成器1292はコンポーネントカウンタ1340とデータバイトカウンタ1341の出力を、ストレージデバイス1293を含むメモリーモジュールの内部読み込みアドレスを計算するために用いている。コンポーネントブロックカウンタ1340はストレージデバイスに格納されている、ピクセルデータブロックから抽出したコンポーネントブロック数を生成する。そのブロック数はデータバイトカウンタ1341の出力を4倍することで与えられる。具体的には、ピクセルブロック分解モードにおける内部読み込みアドレス1348、1349、1350、1351は次のように計算される。コンポーネントブロックカウンタはオフセット値1343、1344、1345、1347を計算するために使用され、また出力データバイトカウンタ1341はベース読み込みアドレス1354を生成するために用いられる。オフセット値1343はベース読み込みアドレス1354に加算された1358で、加算値は内部読み込みアドレス1348（あるいは1349、1350、1351）である。メモリモジュールのオフセット値は、多重メモリモジュールで実行される同時読み込みに対して一般的に異なる値をとるが、コンポーネントブロックの抽出においては本質的に同じである。ピクセルデータブロック分解モードにおける4つの内部読み込みアドレスを計算するのに用いられるベースアドレス1354も同様である。インクリメント信号1308はコンポーネント

バイトカウンタのインクリメント信号として使用される。カウンタは読み込みが成功する度にインクリメントされる。コンポーネントブロックカウンタインクリメント信号1356は、単一校正用をデータブロックが正常にバッファから取り出された後、コンポーネントブロックカウンタ1340をインクリメントするのに用いられる。

【0342】出力ローテーション制御信号1304（図116）はコンポーネントブロックカウンタの出力と出力データバイトカウンタの出力から取り出され、本質的に内部アドレスの生成と同じ方法である。コンポーネントブロックカウンタの出力はローテーションオフセット1347を計算するのに用いられる。出力ローテーション制御信号1304はローテーションオフセット1355とベース読み込みアドレス1354の和の最下位2ビットにより与えられる。入力ローテーション制御信号は、アドレス及びローテーション制御信号生成器の例の様に、外部書き込みアドレス1305の最下位2ビットにより与えられる。

【0343】図120は、再コンフィギュレーション可能なMUVバッファ250に格納された単一コンポーネントデータからの多重チャネルピクセルデータの再構成に用いられるもう1つのアドレス生成器1292である。この場合、バッファはJPEG伸長のためのJPEGモード（モード2）となる。この場合、単一コンポーネントデータブロックはバッファに格納され、ピクセルデータブロックはバッファから取り出される。この例では、メモリモジュールへの書き込みアドレスは、本質的な変更なしで外部書き込みアドレス1305によって与えられる。単一コンポーネントブロックは連続したメモリに格納される。この例の入力ローテーション制御信号1303は単に書き込みアドレスの最下位2ビットによってセットされる。ピクセルカウンタ1360は、バッファ内に格納されている単一コンポーネントブロックから抽出されたピクセル数の記録を保持するために用いられる。ピクセルカウンタの出力は、読み込みアドレス1348、1349、1350、1351及び出力ローテーション制御信号1304を生成するために用いられる。一般に読み込みアドレスは、ストレージデバイス1293を構成するそれぞれのモジュール毎に異なっている。この例では、読み込みアドレスは単一コンポーネントブロックインデックス1362、1363、1364、1365あるいは1365とバイトインデックス1361の2つの部分からなる。特定のブロックの単一コンポーネントブロックインデックスを計算するために、オフセットが出力ピクセルカウンタのビット3と4に加えられる。一般にオフセット1366、1367、1368、1369はそれぞれの読み込みアドレスで異なる。ピクセルカウンタのビット2からビット0は読み込みアドレスのバイトインデックス1361に用いられる。読み込

みアドレスは図120に示されるように、単一コンポーネントブロックインデックス1362、1363、1364、1365あるいは1365とバイトインデックス1361の結合の結果である。この例では、出力ローテーション制御信号1304は、本質的な変化なしにピクセルカウンタの出力のビット4とビット3により生成される。インクリメント信号1308はピクセルカウンタ1360をインクリメントするためのピクセルカウンタインクリメント信号として使用される。ピクセルカウンタ1360はピクセルが正常にバッファから取り出されたときにインクリメントされる。

【0344】図121はストレージデバイス1293の構造である。ストレージデバイス1293は1383、1384、1385の3つの4ビットワイドメモリモジュールと1380、1381、1382の3つの8ビットワイドメモリモジュールを持つことができる。メモリモジュールは単一ルックアップテーブルモード（モード0）の36ビットのワード、多重ルックアップテーブルモード（モード1）の12×3ビットのワード、JPEGモード（モード2）における32ビットのピクセルあるいは4×8ビットの単一コンポーネントデータを格納するために結合できる。通常それぞれのメモリモジュールは符号化された入力及び出力データストリーム（1296と1297）の異なる部分に関連づけられる。たとえば、メモリモジュール1380は符号化された入力データストリーム1296のビット0からビット7に接続されデータ入力ポートと符号化された出力データストリーム1297のビット0からビット7に接続されたデータ出力ポートをもつ。この例ですべてのメモリモジュールの書き込みアドレスは一緒に接続され、同時に同じ値を共有する。一方、図121に示されるメモリモジュールの読み込みアドレス1386、1387、1388、1390、1391は読み込みアドレス生成器1292により与えられ、これらは一般に異なる値をとる。例では、共通の書き込み有効信号はすべての8ビットメモリモジュールに対して書き込み有効信号を出すために用いられ、第二の共通の書き込み有効信号はすべての4ビットメモリモジュールに対して書き込み有効信号を出すために用いられる。

【0345】図122はストレージデバイス1293内のメモリモジュールにアクセスするための読み込みアドレス1386、1387、1388、1389、1390を生成するための組み合わせ回路の回路図である。符号化されたそれぞれの入力データオブジェクトは部分部分に分解され、それぞれの部分はストレージデバイスの独立したメモリモジュール内に格納される。従って通常、すべての処理モードにおけるすべてのメモリモジュールの書き込みアドレスは本質的には同じであり、メモリモジュールの書き込みアドレスを計算するために実質的にロジックは必要ない。一方、読み込みアドレスは通

常、処理毎に異なり、それぞれの処理モードにおけるメモリモジュールそれぞれに対しても異なる。再コンフィギュレーション可能なMUVバッファ250の出力データストリーム1298内のすべてのバイトはJPEG圧縮のJPEGモード（モード2）のバッファに格納されているピクセルデータから抽出された単位コンポーネントデータ、あるいはJPEG伸長のJPEGモードのバッファ内に格納されて単一コンポーネントデータから抽出されたピクセルデータを含まなくてはならない。出力データに対する要求はバッファへの4つの読み込みアドレス1348、1349、1350、1351の生成によって満たされる。多重ルックアップテーブルモード（モード1）においては、最大3つの検索テーブルがバッファに格納され、従って最大3つまでの読み込みアドレス1348、1349、1350が3つの検索テーブルにインデックスをつけるために必要である。すべてのメモリモジュールの読み込みアドレスは単一ルックアップテーブルモード（モード0）の場合と同じであり、読み込みアドレス248のみがこのモードで用いられる。図122に示されている制御回路の例はストレージデバイス1293を構成する6つのメモリモジュールそれぞれの読み込みアドレス1386-1391を計算するために、バッファの処理モード信号と最大4つの読み込みアドレスを用いる。読み込みアドレス生成器1292は入力信号として外部アドレスバス1348、1349、1350、1351からなる外部読み込み信号をもちい、ストレージデバイス1293を構成するメモリモジュールの内部読み込みアドレス1386、1387、1389、1390を生成する。

【0346】図123はバッファ250が単一ルックアップテーブルモードにある時に、どのようにして20ビットの行列係数がバッファ250に格納されるのかを示した図である。この場合、データオブジェクトが再コンフィギュレーション可能なMUVバッファに書き込まれるときにはキャッシュ上のデータオブジェクトに対してエンコーディングは通常行われない。行列係数は8ビットメモリモジュール1380、1381、1382に格納される。行列係数のビット7からビット0はメモリモジュール1380に格納され、ビット15からビット8はメモリモジュール1381に格納され、ビット19からビット16はメモリモジュール1382の下位4ビットに格納される。命令の残りのために必要であるようなバッファに格納されたデータオブジェクトは何回も取り出される。単一ルックアップテーブルモードにおける、すべてのメモリモジュールの読み込みと書き込みのアドレスは本質的に同じである。

【0347】図124は多重ルックアップテーブルモード（モード1）において、どのようにしてバッファにテーブルエントリが格納されるのかを示した図である。この場合、3つの検索テーブルはバッファに格納され、それ

ぞれの検索テーブルは4ビットのインターバル値と8ビットの小数値をもつ。通常インターバル値は4ビットのメモリモジュールに格納され、小数値は8ビットのメモリモジュールに格納される。この場合3つの検索テーブル1410、1411、1412はメモリバンク1380と1383、1381と1384、1382と1385に格納される。分離過去も未有効制御信号1306と1307(図121)はストレージデバイスに格納されている小数値に影響せずにストレージデバイス1293にインターバル値を書き込むことができる。本質的に同様な方法でインターバル値に影響を与えずに小数値を書き込むことができる。

【0348】図125はピクセルデータブロックを単一要素データブロックに分解するJPEGモード(モード2)の状態の再コンフィギュレーション可能なMUVバッファ250にどのようにしてピクセルデータが書き込まれるのかを示した図である。ストレージデバイス1293は、8ビットメモリモジュールと同様な方法で統合して扱われるメモリモジュール、1381と1384を含むメモリモジュール1380、1381、1382、1383、1384からなる4つの8ビットメモリバンクとして統括される。メモリモジュール1385はJPEGモード(モード2)では使用されない。32ビットの符号化されたピクセルは4つのバイトに分解され、それぞれが異なる8ビットのメモリモジュールに格納される。

【0349】図126は単一コンポーネントモードであるストレージデバイス1293にどのようにして単一コンポーネントデータブロックが格納されるのかを示した図である。ストレージデバイス1293は、8ビットメモリモジュールと同様な方法で統合して扱われるメモリモジュール、1381と1384を含むメモリモジュール1380、1381、1382、1383、1384からなる4つの8ビットメモリバンクとして統括される。メモリモジュール1385はJPEGモード(モード2)では使用されない。32ビットの符号化されたピクセルは4つのバイトに分解され、それぞれが異なる8ビットのメモリモジュールに格納される。この場合、単一コンポーネントブロックは64バイトからなる。単一のコンポーネントブロックが垂直バッファに書き込まれるときは、それぞれに異なる量のバイトローテーションが適用される。32ビットの符号化されたピクセルデータはバッファ内の異なる単一コンポーネントデータブロックを読むことで取り出される。

【0350】より詳細な再コンフィギュレーション可能なデータバッファ250の統括方法は、ピクセルオーガナイザの節を参照せよ。以上の具体例では、再コンフィギュレーション可能なデータバッファが、異なる命令と関係するデータの処理に用いられることを示した。3つの処理モードのある再コンフィギュレーション可能なデ

ータバッファが明らかにされた。異なるアドレスの生成技術がバッファの処理モードのそれぞれにおいて必要となる。単一ルックアップテーブルモード(モード0)は画像変換において、行列係数をバッファに格納するのに用いられる。多重ルックアップテーブルモード(モード1)では多チャンネルの色空間変換(CSC)における多数のインターバル及びフラクション検索テーブルをバッファに格納するのに用いられる。JPEGモード(モード2)はJPEG圧縮、JPEG伸長それぞれにおいて、MCUデータを8×8の単一コンポーネントブロックに分解、あるいは8×8の単一コンポーネントブロックをMCUに再合成するのに用いられる。

【0351】3.18.3 結果オーガナイザ
MUVバッファ250は結果オーガナイザ249においても用いられる。結果オーガナイザ249は、メインデータパス242あるいはJPEGコード241のストリームをバッファしてフォーマットする。結果オーガナイザ249はまた、図42で説明した結果データの圧縮、非圧縮、非正規化、バイトレインスワップ、再編成にも関係する。更に結果オーガナイザ249は外部インターフェースコントローラ238、ローカルメモリコントローラ236、周辺インターフェースコントローラ237の要求に対し、その結果を転送する。

【0352】JPEG伸長モードの時、結果オーガナイザ249はMUV RAM250をJPEGコード249の画像データをダブルバッファするために用いる。ダブルバッファはMUV RAM250の半分には書き込まれているJPEGコード241のデータを用いてJPEG伸長する場合に、同時に残りの半分には書きこまれた画像データが指定の格納場所へ出力されるとき、そのパフォーマンスをあげることができる。

【0353】1、3及び4チャンネル画像データは、同一チャンネルからの8ビットのコンポーネントを含む8×8ブロックの形のJPEG伸長を行っている間に、結果オーガナイザ249に渡される。結果オーガナイザはこれらのブロックを指定の順番でMUV RAM250に格納し、また複数チャンネルのインターリーブ画像のために、データをMUV RAM250から読みこみを行っている時のチャンネルのメッシュを格納する。例えば、YUVによる3チャンネルのJPEG圧縮ではJPEGコード241は3つの8×8ブロックを、初めにY、次にU、最後にVの順で出力する。メッシュ処理はそれぞれブロックか1つのコンポーネントを取り出すことによって行われ、ピクセルを(YUVX)の形で構成する。ここでXは未使用チャンネルである。バイトスワッピングは出力チャンネルのスイッチが必要となったときに行われる。結果オーガナイザはまた、伸長された出力データのクロマデータの再構成のための必要なサブサンプリング処理を行う必要がある。このことは生成するためにそれぞれのプログラムチャンネルを繰り返すという意味を含んでいる。

【0354】図127にもどると図2の結果オーガナイザ249の詳細が示されている。結果オーガナイザ249は、その処理に設定されるレジスタのレジスタファイルを含む通常の標準Cbusインターフェース840周辺に基礎をおいている。結果オーガナイザ249の処理はピクセルオーガナイザ249と同様であるが、リバーデータ操作が行われる。データ操作ユニット842はバイトレインスワッピング、コンポーネント代入、コンポーネント解放、非正規化をMUVアドレス発生器805により生成されるデータに対して行う。実行された処理は図42を参照して前述の通り説明され、内部レジスタにセットされた様々なフィールドに従って処理が行われる。FIFOキュー843は出力データをそれがRbus制御ユニット844を用いて出力される前にバッファを行う。Rbus制御ユニット844はアドレスデコーダとアドレス生成器によって構成される。格納モジュール用のアドレスは、必要な出力バイト数のデータに加えて、内部レジスタに格納される。更に、内部RO_CUTレジスタはいくつくらいの出力バイトが出力バスのバイトストリーム上に送られる前に欠落したかを決定する。加えて、RO_LMTレジスタは出力制限が中止された後の次のデータを用いて最大いくつのデータ項目が出力されるかを決定する。MAG805はJPEG伸長時にMUVRAM250のアドレスを生成する。MUVRAM250はJPEGコードからの出力をダブルバッファするために用いられる。MAG805は内部コンフィギュレーションレジスタに依存するMUVRAM250におけるコンポーネントのメッシュを行い、ピクセルの入った単一チャンネル、3チャンネル、4チャンネルの出力を行う。バイトレインスワッピングがピクセルデータを適切な場所に格納する前に必要となるので、MUVRAM250から得られるデータはデータ操作ユニットを通して渡される。結果オーガナイザ249がJPEGモードになっていないときはMAG805は単にPbusレシーバ845のデータをデータ操作ユニット842にダイレクトに送る。

【0355】3.18.4 オペランドオーガナイザB及びC

図2に再び戻って、2つの独立なオペランドオーガナイザ247と248はデータキャッシュコントロール240のデータバッファの機能と、JPEGコード241あるいはメインデータバス242にデータを転送する機能を持つ。オペランドオーガナイザ247と248は様々なモードで操作される。

(a) オペランドオーガナイザがCbus要求にたいしてのみ応答するアイドルモード

(b) 現在の命令のデータがオペランドレジスタの内部レジスタに格納されている時の直接モード

(c) オペレータオーガナイザがシーケンシャルアドレスおよびデータキャッシュコントローラ240のバッ

ファが満杯である時のデータを生成するシーケンシャルモード。

【0356】多数のメインデータバス242の処理モードは、少なくともどちらかのオペランドオーガナイザにシーケンシャルモードであることを要求する。オペランドオーガナイザB247における、合成を含むこれらのモードは、ほかのイメージを用いて合成されるバッファピクセルが必要である。オペランドオーガナイザC248はそれぞれのデータチャンネルの値の減衰を行う合成処理に用いられる。ハーフトーンモードではオペランドオーガナイザB247は8ビットの行列係数のバッファを行い、階層的画像フォーマット分解モードではオペランドオーガナイザB247は垂直補間と残部融合命令の両方のデータのバッファを行う。

(d) 定常モードではオペランドオーガナイザBは単一の内部データワードの組立とそのワードを内部レジスタによって指定された回数繰返すことを行う。

(e) タイルモードではオペランドオーガナイザBはピクセルタイルを構成するデータのバッファを行う。

(f) ランダムモードでは、オペランドオーガナイザはデータキャッシュコントローラにMDP242あるいはJPEGコード241のアドレスをダイレクトに転送する。

【0357】内部長さレジスタは、シーケンシャル、タイル、定常の各モードの処理の時に、オペランドオーガナイザ247、248の個々で生成される項目の数を決定する。オペランドオーガナイザ247、248それぞれは、はそれまでに処理されたデータ項目の数を保持し、内部レジスタによって決定される値に達したら停止する。オペランドオーガナイザそれぞれは、バイトレインスワッピングを用いた入力データのフォーマット、コンポーネントの代入、圧縮・非圧縮・正規化機能、にたいしてより信頼がある。要求された処理は内部レジスタを用いてコンフィギュレーションされる。更に、オペランドオーガナイザ247と248それぞれはデータ項目を制限するためにコンフィギュレーションされる。

【0358】図128では、オペランドオーガナイザ(247、248)のより詳細な構成が示されている。

オペランドオーガナイザ247、248は通常の標準Cbusインターフェースとオペランドオーガナイザ全体の制御を司るレジスタ850を含む。更に、Obus制御ユニット851はデータキャッシュコントローラに接続され、シーケンシャル、タイル、定常の各モードのアドレス生成、オペランドオーガナイザ247、248のObusインターフェースの通信を可能にする制御信号の生成、入力ストリームの過去のクロックサイクルから保存される状態を必要とする、正規化、繰返し等を行うデータ操作ユニットの制御を行う。オペランドオーガナイザ247、248がシーケンシャル、あるいはタイルモードであるときには、Obusコントローラユニッ

ト851はデータの要求をデータキャッシュコントローラに送る。このときアドレスは内部レジスタによって決定されている。

【0359】それぞれのオペランドオーガナイザは更に、様々なモードの処理において、データキャッシュコントローラ240からのデータをバッファするために用いられる36ビット幅のFIFOバッファ852を含んでいる。データ操作ユニット853は、ピクセルオーガナイザ246のデータ操作ユニット804に対応する機能と同じ機能を行う。

【0360】メインデータバス/JPEGコーデインタフェース854は通常処理モードにおいてメインデータバスやJPEGコーデモジュール242、241でやりとりされるデータ及びアドレスを分配する。MDP/JCインタフェース854はデータ操作ユニット853からのデータをメインデータバス及びそのデータを繰り返すように構成されたプロセスに送る。色変換モードの場合には、ユニット851、854はデータキャッシュコントローラ240と色変換テーブルの高速アクセスを確立するためにバイパスされる。

【0361】3.18.5 主データバス部
以下の実施例の特徴は、複数の画像処理動作を高速で行うことのできる低価格のコンピュータアーキテクチャを提供する画像プロセッサに関するものである。更に、画像プロセッサは、元々は規定されなかった画像処理動作を行うように構成されることのできる、柔軟性のあるコンピュータアーキテクチャを提供することを目的とする。また、画像プロセッサは、同じロジックをたくさん持っていて、設計プロセスが簡単で安くなるような、コンピュータアーキテクチャを提供することをも目的とする。

【0362】コンピュータアーキテクチャは制御レジスタブロック、復号ブロック、データオブジェクトプロセッサ、および、フロー制御ロジックを具備する。制御レジスタブロックは画像処理動作に関する全ての情報を格納する。復号ブロックは情報を構成信号に復号し、入力データオブジェクトインタフェースを構成する。入力データオブジェクトインタフェースはデータオブジェクトを外部から受け取り格納する。そして、これらのデータオブジェクトをデータオブジェクトプロセッサに配分する。ある画像処理動作においては、入力データオブジェクトインタフェースがデータオブジェクトのアドレスを生成することもあり、これらのデータオブジェクトのソースが正しいデータオブジェクトを提供できるようになる。データオブジェクトプロセッサは、受け取ったデータオブジェクトに対して算術演算を行う。フロー制御ロジックは、データオブジェクト処理ロジックの中のデータオブジェクトフローを制御する。

【0363】特に、データオブジェクトプロセッサは、いくつかの同一なデータオブジェクトサブプロセッサを

備えることができ、各サブプロセッサは、入力データオブジェクトの一部を処理する。データオブジェクトサブプロセッサは、データオブジェクトの当該部分に対し算術演算を行ういくつかの同一な多機能算術部、出力データオブジェクトを処理する後処理ロジック、および、多機能算術部と後処理部とを接続する多重化ロジックを有する。多機能算術部は計算されたデータオブジェクトのための記憶装置を具備する。この記憶装置は、フロー制御ロジックによってイネーブルされるか又はデスエーブルされる。多機能算術部および多重化ロジックは、復号ロジックによって生成された構成信号によって構成される。

【0364】なお、復号ロジックからの構成信号は外部プログラミングエージェントによって変化されることができる。このメカニズムを通じて、どのような多機能ブロックおよび多重化ロジックであっても、外部プログラミングエージェントによって個々に構成することができ、前もって規定されなかった画像処理動作を行うように画像プロセッサを構成することを可能にする。本発明の実施例が持つこれらの特徴およびその他の特徴を以下で詳述する。

【0365】図2において、前述したように、主データバス部242はJPEGデータ符号化以外の全てのデータ操作動作および命令を行う。これらの命令には、合成、色空間変換、画像変換、畳込み演算、行列の乗算、ハーフトーン処理、メモリ複写、および階層画像フォーマットの解凍が含まれる。主データバス242はピクセルオーガナイザ246およびオペランドオーガナイザ247、248から、ピクセルとオペランドデータとを受け取り、結果出力を結果オーガナイザ249に送る。

【0366】図129は、主データバス部242のブロック図である。主データバス部242は汎用の画像プロセッサであって、入力インタフェース1460、画像データプロセッサ1462、命令ワードレジスタ1464、命令ワード復号器1468、制御信号レジスタ1470、レジスタファイル1472、および、ROM1475を備える。

【0367】命令制御部235は、バス1454を通じて、命令ワードを命令ワードレジスタ1464へ移す。それぞれの命令ワードは、実行すべき画像処理動作の種類や画像処理動作の様々なオプションを選択するプラグなどの情報を含んでいる。命令ワードは、バス1465を経由して命令ワード復号器1468に運ばれる。それで、命令制御部235は、命令ワードを復号するように命令ワード復号器1468に指示することができる。その指示を受けると、命令復号器1468は命令ワードを制御信号に復号する。それから、これらの制御信号はバス1469を経由して制御信号レジスタ1470に運ばれる。それから、制御信号レジスタの出力は、バス1471を経由して入力インタフェース1460および画

像データプロセッサ1462に接続される。

【0368】主データバス部242をより柔軟性のあるものにするために、命令制御部235が制御信号レジスタ1470に直接書き込むこともできる。これによって、主データバス部242の構造を熟知している誰でも、主データバス部242の細かい構成を行えるようになり、主データバス部242は命令ワードで記述されていない画像処理動作をも実行できるようになる。

【0369】所望の画像処理動作を実行するために必要な全ての情報を命令ワードに収容できない場合、命令制御部235は、その収容できない必要な全ての情報をレジスタファイル1472のいくつかの選ばれたレジスタに書き込むことができる。この情報は、バス1473を経由して、入力インターフェース1460および画像データプロセッサ1462に伝えられる。ある画像処理動作において、入力インターフェース1460は主データバス部242の現在状態を反映するために、レジスタファイル1472の選ばれたレジスタの内容を更新することもあり得る。画像処理動作を実行するときに問題が生じたとき、命令制御部235は前述の特徴を用いて、問題を容易に発見できるようになる。

【0370】命令ワードの復号が終了し、制御信号レジスタに所望する制御信号がロードされたとき、命令制御部235は主データバス部242に所望画像処理動作の実行を始めるように指示することができる。この指示を受けると、入力インターフェース1460はバス1451からのデータオブジェクトを受け取り始める。入力インターフェース1460は、実行される画像処理動作の種類に応じて、オペランドバス1452又はオペランドバス1453からのオペランドデータを受け取り始めるか、或は、オペランドデータのアドレスを生成してオペランドバス1452又はオペランドバス1453からのオペランドデータを受け取り始める。入力インターフェース1460は、制御信号レジスタ1470の出力に応じて、入力データを格納して配列し直す。アフィン画像変換動作および畳込み演算のような計算を行うとき、入力インターフェース1460はバス1452および1453を経由してフェッチされるべき座標をも生成する。

【0371】画像データプロセッサ1462は、入力インターフェース1460に配列し直してもらったデータオブジェクトに対して主算術演算を行う。画像プロセッサ1462は、所定の補間ファクタで行われる2つのデータオブジェクトの間の補間、2つのデータオブジェクトの乗算、及びその結果を255で割る割算、2つのデータオブジェクトに対する通常の乗算及び足し算、データオブジェクトの分数部に対する様々な精度での切り捨て、データオブジェクトのオーバフローをある最大値に、そしてデータオブジェクトのアンダフローをある最低値にそれぞれ抑えるクランプ、データオブジェクトのスケーリング及びクランピングというような処理を行う

ことができる。バス1471の制御信号は、前記の算術演算中のどれがデータオブジェクトに対して行われるか、及びその動作の順序などを制御する。

【0372】ROM1475は、8.8フォーマットで切り捨てられた255/xの被除数を有するが、ここで、xは0から255までの数である。ROM1475は、バス1476を経由して、入力インターフェース1460および画像データプロセッサ1462に接続される。ROM1475は短い長さのブレンドを生成し、データオブジェクトに255を掛け、その結果を他のデータオブジェクトで割るというような動作に用いられる。

【0373】オペランドバス、例えば1452の数は2に制限されるが、大多数の画像処理動作においては十分である。図130は、入力インターフェース1460をより詳細に示す。入力インターフェース1460は、データオブジェクトインターフェース部1480、オペランドインターフェース部1482および1484、アドレス生成状態器1486、ブレンド生成状態器1488、行列乗算状態器1490、補間状態器1494、データ同期部1500、算術部1496、他レジスタ1498、並びに、データ分配ロジック1505を備える。

【0374】データオブジェクトインターフェース部1480と、オペランドインターフェース部1482及び1484とは、外部からデータオブジェクト及びオペランドを受け取る。インターフェース部1482、1484は、2つとも制御バス1515からの制御信号によって構成される。インターフェース部1482、1484は、受け取ったばかりのデータオブジェクト/オペランドを含むデータレジスタを内部に有しており、2つとも前記データレジスタが有効であるときはVALID信号を出力する。インターフェース部1482、1484のデータレジスタの出力はデータバス1505に接続される。インターフェース部1482、1484のVALID信号はフローバス1510に接続される。オペランドをフェッチするように構成されたとき、オペランドインターフェース部1482および1484は、算術部1496からのアドレスと、行列乗算状態器1490と、データオブジェクトインターフェース部1480のデータレジスタの出力とを受け取り、その中で必要なアドレスを制御バス1515からの制御信号に応じて選択する。いくつかの場合、特に、外部からデータを受けて格納する必要がない場合、オペランドインターフェース部1482および1484のデータレジスタは、データオブジェクトインターフェース部1480または算術部1496のデータレジスタの出力からデータを格納するように構成されることが有り得る。

【0375】アドレス生成状態器1486は、アフィン画像変換動作および畳込み演算動作において算術部1496を制御し、ソース画像のアクセスされるべき次の座標を計算する。アドレス生成状態器1486は、制御バ

ス1515のSTART信号が設定されることを待つ。制御バス1515のSTART信号が設定されると、アドレス生成状態器1486はデータオブジェクトインターフェース部1480に対してSTALL信号を解除して、データオブジェクトが到着することを待つ。なお、アドレス生成状態器1486は、アドレス生成状態器1486がフェッチすることを必要とするカーネルデスクリプタのデータオブジェクトの数と同じとなるようにカウンタを設定する。カウンタの出力は、復号され、オペランドインターフェース部1482および1484のデータレジスタと他レジスタ1498とのイネーブル信号になる。データオブジェクトインターフェース部1480からVALID信号が起動されると、アドレス生成状態器1486はカウンタを減少させるようになり、データオブジェクトの次の部分が異なるレジスタにラッチされる。

【0376】カウンタが零に達すると、アドレス生成状態器1486はオペランドインターフェース部1484からインデックステーブル値とピクセルとをフェッチし始めよとオペランドインターフェース部1482に指示する。なお、アドレス生成状態器1486は、行の数と列の数とをそれぞれ持つ2つのカウンタをロードする。全てのクロックエッジにおいて、かつオペランドインターフェース部1482などからのSTALL信号によって停止されないとき、カウンタは減少され残りの行と列を出力する。そして、算術部1496は、フェッチされるべき次の座標を計算する。両方のカウンタが零に達すると、カウンタは行と列の数を再びロードし、算術部1496は次の行列の左上端を探すように構成される。

【0377】ピクセルの真の値を決定するために補間が使われる場合、アドレス生成状態器1486は2つのクロックサイクルごとに、行および列の数を減少させる。これは1ビットカウンタを使って実行され、その出力は行および列カウンタのイネーブルとして用いられる。行列が一度スキャンされた後、状態器は長さカウンタのカウンタを減少させる信号を送る。カウンタが1に達して、かつ最終インデックステーブルアドレスがオペランドインターフェース部1482に送られたとき、状態器は最終信号を出し、開始ビットをリセットする。

【0378】ブレンド生成状態器1488は、算術部1496を制御して、ブレンド長さのための0から255までの数列を生成する。この数列は、ブレンド開始値とブレンド終了値との間を補間する補間ファクタとして使われる。ブレンド生成状態器1488はどちらかのモード（ジャンプモード又はステップモード）で実行すべきであるかを決める。ブレンド長さが256以下である場合はジャンプモードが使われ、そうでない場合はステップモードが使われる。

【0379】ブレンド生成状態器1488は、下記の計算を行い、その結果をレジスタ（reg0, reg1,

reg2）にセットする。ブランドランプが予め決定された長さでステップモードにある場合、511-長さをreg0（24ビット）に、512-2*長さをreg1（24ビット）に、そして、終了-開始をreg2（4×9ビット）に、それぞれラッチする。ランプがジャンプモードにある場合は、0をreg0（24ビット）に、255/（長さ-1）をreg1（24ビット）に、そして、終了-開始をreg2（4×9ビット）に、それぞれラッチする。

【0380】ステップモードにおいて、以下の処理が各サイクルにおいて実行される。reg0>0であるとき、reg0にreg1を加え、その結果をreg0に格納する。もう一つのインクリメンタがイネーブルされることもできるが、その場合には出力が1だけ増加される。reg0≤0であるとき、reg0に510を加え、その結果をreg0に格納する。インクリメンタは増加されない。インクリメンタの出力はランプ値である。

【0381】ジャンプモードにおいて、以下の処理が各サイクルにおいて実行される。reg0にreg1を加える。加算の出力は24ビットであり、16.8の固定少数点フォーマットで出力される。前記加算出力をreg0に格納する。分数結果の第1ビットが1である場合、整数部を増加させる。インクリメンタの整数部の下位8ビットはランプ値である。このランプ値、即ちreg2の出力と、ブレンド開始値とは画像データプロセッサ1462に送られ、ランプを生成する。

【0382】行列乗算状態器1490は、変換行列を用いて入力データオブジェクトに対する線形色空間変換を行う。変換行列は4×5次元である。第1から第4列にはデータオブジェクトの4チャネルを掛けるようになっており、最後列は積の和に加えられるべき常係数を含んでいる。制御バス1515からのSTART信号が起動されたとき、行列乗算状態器は以下のように動く。

【0383】1) バス1482及び1484から変換行列の常係数をフェッチすべきライン番号を生成する。なお、他レジスタ1498をイネーブルして常係数が格納できるようにする。

2) 1ビットフリップフロップを備えていて、ライン番号を生成して、バス1482および1484から行列の半分をフェッチするときにアドレスとして使う。なお、データオブジェクトの半分から、前記行列の半分に掛けられるべきものを選択する“MAT_SEL”信号をも生成する。

【0384】3) データオブジェクトインターフェース部1480から入力されるデータオブジェクトがないとき終了する。

補間状態器1494は、データオブジェクトの水平補間を行う。水平補間において、主データバス部242はバス1451からデータオブジェクトストリームを受け取

り、隣のデータオブジェクトの間を補間する。そして、元ストリームの2倍、又は4倍の長さであるデータオブジェクトのストリームを出力する。データオブジェクトはバイト又はピクセルにパックされることがあり得るため、補間状態器1494は、スループットが最大になるようにそれぞれの場合に異なる操作を行う。補間状態器1494は以下のように動作する。

【0385】1) INT_SEL信号を生成することによって、データ配分ロジック1503が入力データオブジェクトを再配列するようにし、正しいデータオブジェクト対に対して補間を行うようにする。

2) 隣接するデータオブジェクト対の間を補間するための補間ファクタを生成する。

【0386】3) データオブジェクトインターフェース部1480がもうデータオブジェクトを受け入れないようにするSTALL信号を生成する。これが必要とされる理由は、出力ストリームが入力ストリームより長いからである。STALL信号はフローバス1510に送られる。

算術部1496は、算術計算を行うための回路を具備しており、制御バス1515の制御信号によって構成される。これは、アフィン画像変換および畳込み演算と合成においてのブレンド生成という2つの命令のみによって使われる。

【0387】アフィン画像変換および畳込み演算において、算術部1496は以下のような演算を行う。

1) 次のxおよびy座標を計算する。x座標を計算するために、算術部1496は加算器を用いて現在のx座標に水平および垂直デルタのx成分を加えるか、減算器を用いて現在のx座標から水平および垂直デルタのx成分を引くようにする。y座標を計算するために、算術部1498は加算器を用いて現在のy座標に水平又は垂直デルタのy成分を加えるか、減算器を用いて現在のy座標から水平又は垂直デルタのy成分を引くようにする。

【0388】2) y座標をインデックステーブルオフセットに加算しインデックステーブルアドレスを計算する。ピクセルの元の値を求めるために補間を使う場合、前記の和はインデックスエントリを求めるために、更に4だけ増加される。

3) x座標をインデックステーブルエントリに加算し、ピクセルのアドレスを求める。

【0389】4) 長さカウンタから1を引く。

ブレンド生成において、算術部1496は以下のように作動する。

1) ステップモードにおいて、ある1つのランプ加算器を用いてランプ生成アルゴリズムの内部変数を計算する。一方、その他の1つの加算器は、インターバル変数が零より大きいときにランプ値を増加させるために用いられる。

【0390】2) ジャンプモードにおいては、ジャンプ

値を現在のランプ値に加えるために1つの加算器のみが必要とされる。

3) ジャンプモードでは、分数の切り捨てが行われる。

4) ランプ生成の始めにあたって、ブランドの終了からブランドの開始を引く。

【0391】5) 長さカウンタから1を引く。

他レジスタ1498は、データオブジェクトインターフェース部1480、並びに、オペランドインターフェース部1482及び1484において、データレジスタ以外の余分の格納空間を提供する。他レジスタ1498は、内部変数を格納するか、或はデータオブジェクトインターフェース部1480からの過去のデータオブジェクトをバッファするのにおいて使われるのが普通である。レジスタ1498は、制御バス1515の制御信号によって構成される。

【0392】データ同期部1500は、制御バス1515の制御信号によって構成される。データ同期部1500は、STALL信号をデータオブジェクトインターフェース部1480、並びに、オペランドインターフェース部1482および1484に提供することによって、あるインターフェース部が、他のインターフェースは持っていない一部データオブジェクトを受け取った場合、他のインターフェースの全てかデータを受け取るまでそのインターフェース部を停止させる。

【0393】データ配分ロジック1505は、行列乗算状態器1490からのMAT_SEL信号と、補間状態器1494からのINT_SEL信号とを含む制御バス1515の制御信号に応じて、データバス1510およびレジスタファイル1472からのデータオブジェクトをバス1530を経由して再配列する。再配列されたデータはバス1461へ出力される。

【0394】図131は、図129の画像データプロセッサ1462をより詳細に示す。画像データプロセッサ1462は、パイプライン制御部1540と、多数のカラーチャネルプロセッサ1545、1550、1555、及び1560とを有する。全てのカラーチャネルプロセッサは、入力インターフェース1460(図131)によって駆動されるバス1565から入力を受け取る。全てのチャネルプロセッサとパイプライン制御部1540は、バス1472を経由する、制御信号レジスタ1470からの制御信号によって構成される。全てのカラーチャネルプロセッサは、図129のレジスタファイル1472及びROM1475からの入力をもバス1580を経由して受け取ることがある。全てのカラーチャネルプロセッサとパイプライン制御部との出力はグループされてバス1570となり、画像データプロセッサ1462の出力1455を形成する。

【0395】パイプライン制御部1540は、全てのカラーチャネルプロセッサのレジスタをイネーブル又はデスエーブルすることによって、全てのカラーチャネル

ロセッサのデータオブジェクトのフローを制御する。パイプライン制御部1540の中には、レジスタパイプラインがある。パイプラインの形態及び長さは、バス1471からの制御信号により構成されるようになっており、パイプライン制御部1540のパイプラインとカラーチャネルプロセッサのパイプラインとは、その形態が同じである。パイプライン制御部はバス1565からVALID信号を受け取る。パイプライン制御部1540のパイプラインステージそれぞれにおいて、入力VALID信号が起動され、パイプラインステージが停止されていない場合、パイプラインステージは全てのカラーチャネルプロセッサに対してレジスタイネーブル信号を起動させるとともに入力VALID信号をラッチする。それから、ラッチの出力、即ち、VALID信号は、次のパイプラインステージに移る。このようにして、パイプラインにおけるデータオブジェクトの移動が、データ記憶装置を用いずに、シミュレートかつ制御される。

【0396】カラーチャネルプロセッサ1545、1550、1555、及び1560は、入力データオブジェクトに対する主な算術動作を行い、各プロセッサは出力データオブジェクトの1つのチャネルを担当している。好適な実施例においては、大多数のピクセルデータオブジェクトが最大4つのチャネルを持っているため、カラーチャネルプロセッサの数は4に制限される。

【0397】カラーチャネルプロセッサの中には、ピクセルの不透明(opacity)チャネルを処理する部分がある。図131には示されていないが、制御バス1471に接続されている追加の回路があり、カラーチャネルプロセッサは不透明チャネルを正しく処理するように制御バス1471からの制御信号を変換する。これは、ある画像処理動作においては、不透明チャネルに対する動作がカラーチャネルに対する動作と少し異なるからである。

【0398】図132は、カラーチャネルプロセッサ1545、1550、1555、1560を(図132においては一般的に1600で示した)より詳細に示す。各カラーチャネルプロセッサ1545、1550、1555、1560は、処理ブロックA1610と、処理ブロックB1615と、ビッグ加算器1620と、分数切り捨て部1625と、クランプまたはラッパー1630と、出力多重化部1635とを備えている。カラーチャネルプロセッサ1600は、制御信号レジスタ1470からの制御信号をバス1602を経由して、パイプライン制御部1540からのイネーブル信号をバス1604を経由して、レジスタファイル1472からの情報をバス1605を経由して、その他カラーチャネルプロセッサからのデータオブジェクトをバス1603を経由して、入力インターフェース1460からのデータオブジェクトをバス1601を経由して、それぞれ受け取る。

【0399】処理ブロックA1610は、バス1601

からのデータオブジェクトに対していくつかの算術動作を行い、部分的に計算されたデータオブジェクトをバス1611に出力する。処理ブロックA1610が画像処理動作のために行うべき処理を以下に説明する。合成において、処理ブロックA1610はデータオブジェクトバス1451からのデータオブジェクトに不透明度を掛け、ブレンド開始値とブレンド終了値との間を図129の入力インターフェース1460からの補間ファクタによって補間し、図129のオペランドバス1452からのオペランドをブレ乗算するかまたはブレンドカラーに不透明度を掛けるかする。そして、ブレ乗算されたオペランドまたはブレンドカラーデータに対する乗算を減衰させる。

【0400】一般色空間変換において、処理ブロックA1610は、図129のバス1451からの2つの分数値を用いて4つのカラーテーブル値の間を補間する。アフィン画像変換および畳込み演算において、処理ブロックA1610はソースピクセルの色に不透明度をブレ乗算し、現在x座標の分数部を用いて同じ行のピクセルの間を補間する。

【0401】線形色空間変換において、処理ブロックA1610はソースピクセルのカラーに不透明度をブレ乗算し、ブレ乗算されたカラーデータに変換行列係数を掛ける。水平補間と垂直補間において、処理ブロックA1610は2つのデータオブジェクトの間を補間する。

【0402】レジデュアルマージンにおいて、処理ブロックA1610は2つのデータオブジェクトを加算する。処理ブロックA1610は多数の多機能ブロック1640と、処理ブロックAグルーロジック1645とを備える。多機能ブロック1640は制御信号によって構成されていて、以下の機能のどちらかの1つを実行することができる。

【0403】2つのデータオブジェクトに対し加減算を行う。1つのデータオブジェクトを伝える。2つのデータオブジェクトの間をある補間ファクタによって補間する。色に不透明度をブレ乗算する。2つのデータオブジェクトを掛け、その積に第3のデータオブジェクトを掛ける。

【0404】2つのデータオブジェクトに対し加減算を行い、その結果に不透明度をブレ乗算する。多機能ブロック1640のレジスタは、図131のパイプライン制御部1540によって生成される、バス1604からのイネーブル信号によってイネーブルされるかデスエーブルされる。処理ブロックAグルーロジック1645はバス1601からのデータオブジェクトおよびバス1603からのデータオブジェクトと、いくつかの多機能ブロック1640の出力とを受け取り、これらをその他の選択された多機能ブロック1640の入力に送る。処理ブロックAグルーロジック1645もバス1602からの制御信号によって構成される。

【0405】処理ブロックB1615は、バス1601からのデータオブジェクトとバス1611からの部分的に計算されたデータオブジェクトとに対して算術動作を行い、部分的に計算されたデータオブジェクトをバス1616に出力する。処理ブロックB1615が画像処理動作のために行う処理を以下に説明する。非正のオペレータをもつ合成において、処理ブロックB1615はデータオブジェクトバス1451からのプレ処理されたデータオブジェクトと、オペランドバス1452からのオペランドに対して、バス1603からの合成被乗数を掛けるとともに、8.8フォーマットの255/不透明度の値であるROMの出力を、クランプ/ラップされたデータオブジェクトに掛ける。

【0406】正のオペレータをもつ合成において、処理ブロックB1615は、プレ処理された2つのデータオブジェクトを加算する。更に、不透明チャンネルにおいては、前記の和から255を引いて、その差をオフセットに掛け、その積を255で割る。一般色空間変換において、処理ブロックB1615は、バス1451からの2つの分数値を用いて4つのカラーテーブル値の間を補間し、残っている分数値を用いて処理ブロックA1610からの部分的に補間されたカラー値と、以前の補間結果との間を補間する。

【0407】アフィン画像変換および畳込み演算において、処理ブロックB1615は、現在y座標の分数部を用いて、部分的に補間されたピクセルの間を補間し、補間されたピクセルにサブサンプルウェイト行列の係数を掛ける。線形色空間変換において、処理ブロックB1615はソースピクセルのカラーに不透明度をプレ乗算し、プレ乗算されたカラーに変換行列係数を掛ける。

【0408】処理ブロックB1615は、多数の多機能ブロックと、処理ブロックBグルーロジック1650とを備える。多機能ブロックは、処理ブロックA1610のものと同様であるが、処理ブロックBグルーロジック1650においては、バス1601、1603、1611、1631からのデータオブジェクトと、選択された多機能ブロックの出力とを受け入れ、これらを選択された多機能ブロックの入力に送る。処理ブロックBグルーロジック1650もバス1602からの制御信号によって構成される。

【0409】ビッグ加算器1620は、処理ブロックA1610と処理ブロックB1615からの部分的結果のいくつかを結合する。これは、バス1601を経由して入力インターフェース1640から、バス1611を経由して処理ブロックA1610から、バス1616を経由して処理ブロックB1615から、そして、バス1605を経由してレジスタファイル1472から、それぞれの入力を受け取り、バス1621に結合された結果を出力する。ビッグ加算器1620も、バス1602の制御信号によって構成される。

【0410】ビッグ加算器1620は、様々な画像処理動作に従って、異なる構成にすることができる。ビッグ加算器1620の所定の画像処理動作における動作を以下に説明する。非正のオペレータを持つ合成において、ビッグ加算器1620は処理ブロックB1615からの2つの部分積を合算する。

【0411】正のオペレータを持つ合成において、オフセットイネーブルが起動されているときに、ビッグ加算器1620は不透明度チャンネルからオフセットのある先処理されたデータオブジェクトの和を引く。アフィン画像変換/畳込み演算において、ビッグ加算器1620は処理ブロックB1615からの積を累算する。

【0412】線形色空間変換において、第1サイクルでビッグ加算器は2つの行列係数/データオブジェクト積と常係数とを合算する。第2サイクルで、直前サイクルの和に他のもう2つの行列係数/データオブジェクト積を加える。分数切り捨て(丸め)部1625は、バス1621を経由してビッグ加算器1620からの入力を受け取り、出力の分数部を切り捨てる。分数部を表すビットの数は、レジスタファイル1472からバス1605のBP信号によって表示される。BP信号を解釈する仕方を以下の表に表す。切り捨てられた出力はバス1626に提供される。

【0413】分数テーブル

【0414】

【表27】

表 27: フラクションテーブル

bp field	Meaning
0	Bottom 26 bits are fractions.
1	Bottom 24 bits are fractions.
2	Bottom 22 bits are fractions.
3	Bottom 20 bits are fractions.
4	Bottom 18 bits are fractions.
5	Bottom 16 bits are fractions.
6	Bottom 14 bits are fractions.
7	Bottom 12 bits are fractions.

【0415】分数切り捨て部1625は、分数の切り捨ての以外に2つの作業を行う。

- 1) 切り捨てられた結果が負であるかどうかを決定する。
- 2) 切り捨てられた結果の絶対値が255より大きいかどうかを決定する。

クランプ又はラッパー1630はバス1626を経由して分数切り捨て部1625から入力を受け取り、下記の動作をその順序に従い行う。

【0416】切り捨てられた結果の絶対値を求めるべきというオプションがイネーブルされているとき、その絶対値を求める。データオブジェクトのアンダフローをある最低値に、そして、データオブジェクトのオーバフローをある最大値に、それぞれクランプする。出力多重化部1635は、バス1616の処理ブロックBの出力と

バス1631のクランプまたはラッパーの出力とのなかで、最終の出力を選択する。なお、データオブジェクトに対して、いくつかの最終処理をも行うが、以下は所定の画像処理動作のために行われる動作を説明する。

【0417】非正のオペレータをもつ、プレ乗算なしの合成において、多重化部1635は処理ブロックB1615のいくつかの出力を結合し、プレ乗算なしのデータオブジェクトを形成する。非正のオペレータをもつ、プレ乗算ありの合成において、多重化部1635はクランプまたはラッパー1630の出力を通過させる。

【0418】正のオペレータをもつ合成において、多重化部1635は処理ブロックB1630のいくつかの出力を結合し、データオブジェクト結果を形成する。一般色空間変換において、多重化部1635は出力データオブジェクトに対して、翻訳・クランプ機能を適用する。他の動作において、多重化部1635は、クランプ又はラッパー1630の出力を通過させる。

【0419】図133は、例えば1640のような、1つの多機能ブロックをより詳細に示す。多機能ブロック1640は、モード検出部1710と、2つの加算オペランド論理部1660及び1670と、3つの多重化論理部1680、1685、及び1690と、2入力加算部1675と、2つの加数を持つ2入力乗算部1695と、レジスタ1705とを備える。

【0420】モード検出部1710は、図129の制御信号レジスタ1470からのMODE信号1711と、図129の入力インターフェース1460からの2つのSUB信号1712及びSWAP信号1713とを受け取る。モード検出部1710は、これらの信号を復号して、加算オペランド論理部1660および1670と、多重化論理部1680、1685、および1690に伝えられる制御信号を生成する。そして、この制御信号は、多機能ブロック1640を種々な動作のできるように構成する。多機能ブロック1640は、8つのモードを有する。

【0421】1) 加減算モード：SUB信号1712に従い、入力1655を入力1665に加えるか、または、入力1665から引く。更に、SWAP信号693に従い、入力をスワップすることもできる。

2) バイパスモード：入力1655を出力にバイパスする。

3) 補間モード：入力1675を補間ファクタとして、入力1655と1665の間を補間する。SWAP信号1713に従い、入力1655および1665をスワップすることができる。

【0422】4) プレ乗算モード：入力1655を入力1675を掛け、その結果を255で割る。INCレジスタ1708の出力は、正しい結果を得るためにバス1707における、このステージの結果を増加すべきかどうかを、次のステージに教える。

5) 乗算モード：入力1655を入力1675を掛ける。

【0423】6) 加減算およびプレ乗算モード：入力1665を入力1655に加えるか、または、入力1655から引き、その結果を入力1675を掛け、そして、この積を255で割る。INCレジスタ1708の出力は、正しい結果を得るためにバス1707にあるこのステージの結果を増加すべきかどうかを、次のステージに教える。

【0424】加算オペランド論理部1660及び1670は、加算器によって減算もできるようにするために、必要に応じて入力に対する1の補数を求める。加算器1675は、バス1662と1672の加算オペランドロジック1660及び1670の出力を合算し、その和をバス1677に出力する。多重化ロジック1680、1685、及び1690は、所望の機能を実行するために適する被乗数と加数を選ぶ。これらは全てモード検出部1710からのバス1714の制御信号によって構成される。

【0425】2つの加数を持つ乗算部1695は、バス1682からの入力をバス1677からの入力に掛ける。そして、前記積にバス1687および1692からの入力の和を加える。加算器1700は、乗算部1695の出力の下位8ビットに乗算部1695の出力の上位8ビットを加える。加算器1700の桁上げはINCレジスタ1701にラッチされる。INCレジスタ1701は、信号1702によってイネーブルされる。レジスタ1705は乗算部1695からの積を記憶する。これも信号1702によってイネーブルされる。

【0426】図134は、合成動作のブロック図を示す。この合成動作は3つの入力データストリームを受け取る。

- 1) 累算ピクセルデータ：この累算部モデルにおいて、結果が格納された位置と同一な位置から誘導される。
- 2) 合成オペランド：カラーと不透明度からなる。カラーと不透明度の両方はフラット、ブレンド、ピクセル、またはタイルであることができる。

【0427】3) 減衰：オペランドデータを減衰する。減衰はフラットなビットマップまたはバイトマップであることができる。

ピクセルデータは典型的に4つのチャネルからなる。その3つのチャネルがピクセルのカラーを形成する。残りのチャネルはピクセルの不透明度である。ピクセルデータはプレ乗算されても、或はされなくてもよい。ピクセルデータがプレ乗算されるとき、各カラーチャネルに不透明度を掛ける。ピクセルがプレ乗算されると合成動作の式が簡単になるため、ピクセルデータがプレ乗算されてから他のピクセルと合成されるのが普通である。

【0428】好適な実施例で実行される合成命令を表1に示す。各命令はプレ乗算されたデータに働きかける。

(ac0, a0) はプレ乗算されたピクセルカラーacと不透明度a0を、rは“オフセット”値、wc()はラップ/クランプ・オペレータを意味し、表1におけるover、in、out、atopの各オペレータの逆オペレータも実装されている。また、合成モデルは左側に累算器を備える。

```

PIXEL Composite (
    IN colorA, colorB:PIXEL;
    IN opacityA, opacityB:PIXEL;
    IN comp_op:COMPOSITE_OPERATOR)
(
    PIXEL result;
    IF comp_opがrover, rin, rout, ratopであるとTHEN
        colorAとcolorBをスワップする;
        opacityA, opacityBをスワップする;
    END IF;
    IF comp_opがover, rover, loado, 又は、plusであるとTHEN
        X=1;
    ELSE IF comp_opがin, rin, atop, 又は、ratopであるとTHEN
        X=opacityB;
    ELSE IF comp_opがout, rout, 又は、xorであるとTHEN
        X=not (opacityB);
    ELSE IF comp_opがloadzero, loadc, 又は、loadcoであるとTHEN
        X=0;
    END IF;
    IF comp_opがover, rover, atop, ratop, 又は、xorであるとTHEN
        Y=not (opacityA);
    ELSE IF comp_opがplus, loadc, 又は、loadcoであるとTHEN
        Y=not (opacityA);
    ELSE IF comp_opがplus, loadc, 又は、loadcoであるとTHEN
        Y=1;
    ELSE IF comp_opがin, rin, out, rout, loadzero, 又は、loado THEN
        Y=0;
    END IF;
    result=colorA * X+colorB * Y;
    RETURN result;

```

命令'load'と'loado'が不透明チャンネルに対して異なる意味を持っているため、以上のコードは不透明サブブロックにおいて異なる。

【0430】図134におけるブロック1765は、ブロック1760の出力をクランプまたはラップする。ブ

【0429】図134における合成ブロック1760は、3つのカラーサブブロックと不透明サブブロックを具備する。各々のカラーサブブロックは、入力ピクセルの1つのカラーチャンネルと不透明チャンネルに対して動作して、出力ピクセルのカラーを得る。以上の動作を擬似コードの形で以下に示す。

ロック1765がクランプするように構成されると、許容される最小値より小さい全ての値を最小値に、許容される最大値より大きい全ての値を最大許容値に抑える。ブロック1765がスワップするように構成されると、以下の式を計算する。

【0431】 $((x - \min) \bmod (\max - \min)) + \min$,

ここで、 \min と \max はカラーにおいて許容される最小値と最大値を意味する。最小値と最大値としては、0と255が望ましい。図134におけるブロック1770は、ブロック1765からの結果をプレ乗算する。これはプレ乗算されたカラー値に $255/o$ を掛けることによりピクセルをプレ乗算する。ここで、 o は合成後の不透明度を意味する。 $255/o$ の値は合成エンジン内のROMから得られる。ROM内の値は8.8フォーマットで記憶されており、分数以下の部分は丸められる。乗算の結果は16.8フォーマットで格納される。逆プレ乗算されたピクセルを生成するために、この結果は8ビットで丸められる。

【0432】ブランド生成部1721は特定の開始値と終了値を持つ特定長さのブランドを生成する。これは以下の2つのステージに渡って行なわれる。

1) ランプ生成

```
Void line draw (length: INTERGER)
{
    d = 511 - length;
    incrE = 510;
    incrNE = 512 - 2 * length;
    ramp = 0;
    for (i = 0; i (length; i++)
    {
        if d (=0 then
            d += incrE;
        else {
            d += incrNE;
            ramp++;
        }
    }
}
```

その後、ランプからブランドを生成するために次の式が使われる。

【0435】

$$\text{Blend} = ((\text{end} - \text{start}) \times \text{ramp} / 255) + \text{start}$$

255による割算に対して切り捨てが行われる。上記式は、2つの加算器と、各チャンネルのランプによって($\text{end} - \text{start}$ に対し)“プレ乗算”を行なうブロックとを必要とする。主データパス部242が行なうことのできる他の画像処理は、一般色空間変換である。一般化色空間変換(GCSC)は出力カラー値を求めるためにピースワイズトライーニア(3次線形)補間を用いる。3次元の入力空間から1次元もしくは4次元出力空間への変換が行なわれるのが望ましい。

【0436】いくつかの場合においては、色域のエッジにおけるトライーニア補間の正確さが問題になる。この問題はエッジ付近に対して敏感なプリントデバイスに

2) 補間

ランプ生成において、合成エンジンは命令の長さに対して、0から255まで線形増加する数列を生成する。ランプ生成には、長さが255以下の“ジャンプ”モードと長さが255より長い“ステップ”モードの2つがある。モードは長さの上位24ビットによって決まる。ジャンプモードにおいて、ランプ値の増加分はクロック周期ごとに少なくとも1である。ステップモードにおいて、ランプ値の増加分はクロック周期ごとに最大1である。

【0433】ジャンプモードにおいて、合成エンジンはステップ値 $255 / (\text{長さ} - 1)$ を求めるために8.8フォーマットのROMを用いる。この値は16ビット累算器に加えられる。累算器の出力は8ビットで切り捨てられて数列を形成する。ステップモードにおいて、合成エンジンはBresenhamの線描アルゴリズムに似たアルゴリズムを用いる。そのアルゴリズムを以下に示す。

【0434】

```
0          if x ≤ 63
out = 2 (x - 64)  if (64 ≤ x ≤ 191)
```

好適な実施例が実行できるその他の画像処理には、画像変換および畳込み演算である。画像変換においてソース画像はスケール、回転、スキューされる。畳込み演算において、ソース画像のピクセルは畳込み行列をもってサ

ンプリングされ、目的画像を生成する。目的画像におけるスキャンラインを生成するためには次の段階が必要である。

【0438】1) 図135に示すような目的画像のスキャンラインを逆変換する。これによって目的画像のスキャンラインを生成するに必要なソース画像のピクセルを識別することができる。

2) ソース画像の必要部分を解凍する。

3) 目的画像の水平、垂直サブサンプリング距離、開始x、y座標をソース画像に逆変換する。

【0439】4) 上記情報を処理部に伝送し、必要なサブサンプリングと補間を行ない、出力画像のピクセルを求める。

サブサンプリング、補間、目的ピクセルの書き込みなどは好適な実施例によって行なわれ、ソース画像における関連する部分、使うべきサブサンプリング周波数などの計算はホストアプリケーションによって行なわれる。

【0440】図136は目的ピクセル値の計算において必要な段階のブロック図である。図136は必要なソース画像のピクセルが利用可能であるものと想定している。目的ピクセルを計算する最後の段階は、ソース画像から2次線形補間された全てのサブサンプルを合算することである。主データパス部242における適当な設定によって引き出される画像変換エンジンのブロック図を図137に示す。画像変換エンジン1830はアドレス生成部1831、プレ乗算部1832、補間部1833、累算部1834、切捨て、クランプ、絶対値を求める論理部1835からなる。

【0441】アドレス生成部1831は、結果ピクセルを構成するのに必要なソース画像のx、y軸を生成する。また、これは入力インデックステーブル1815と画像1810のピクセルからインデックスオフセットを求めるためのアドレスを生成する。アドレス生成部1831がソース画像のx、y軸を生成する前にカーネルディスクリプタを読む。カーネルディスクリプタのフォーマットには2つの種類があり、それを図138に示す。カーネルディスクリプタは、

1) ソース画像の開始座標（符号なしの固定小数点、24.24精度）。位置（0、0）は画像の左上端である。

【0442】2) 水平、垂直のサブサンプルデルタ（2の補数、24.24精度）

3) 固定小数点行列係数における2進小数点の位置を示す3ビットのbpフィールド。図150はbpフィールドの定義とその説明を示す。

4) 累算行列係数。これは20個の2進位置（2の補数）を持つ“可変”小数点精度のものであり、2進小数点の位置はbpフィールドにより暗黙的に規定される。

【0443】5) カーネルディスクリプタのワードの残り個数を示すrlフィールド。この値は行の個数と（列

の個数-1）とを掛けたものと同じである。

短いカーネルディスクリプタにおいて、xの開始座標の定数部を除いた他のパラメータは次のような値を持つ。

xの開始座標の分数 < -0,

yの開始座標 < -0,

水平デルタ < -1.0,

垂直デルタ < -1.0.

アドレス生成部1831が構成された後、現座標を計算する。これにはサブサンプル行列の次元に応じて2つの方法がある。サブサンプル行列の次元が1×1である場合、アドレス生成部1831は十分な座標が得られるまで水平デルタを現座標に加える。

【0444】サブサンプル行列の次元が1×1でない場合、アドレス生成部1831は行列の1つの行が終るまで水平デルタを現座標に加える。その後、アドレス生成部1831は次の行の座標を求めるために垂直デルタを現座標に加える。アドレス生成部1831は次の座標を求めるため、1つ以上の列が終るまで現座標から水平デルタを引く。その後、アドレス生成部1831は垂直デルタを現座標に加え、そしてこの過程を繰り返す。図150の上端におけるダイアグラムは行列へのアクセス方法を示す。この構造を用いて、行列はジグザグでスキャンされ、この方法によって現在のx、y軸が計算されるので、必要なレジスタ数は少なくともよい。累算行列係数はカーネルディスクリプタにおいて同様な順序で並べなければならない。

【0445】現座標を生成した後、アドレス生成部1831はインデックステーブルのアドレスを求めるため、y軸をインデックステーブルベースアドレスに加える

（ソースピクセルが補間されている場合、アドレス生成部1831は次のインデックステーブルも求める必要がある）。インデックステーブルベースアドレスは（y+0）におけるインデックステーブルエントリを指す。インデックステーブルからインデックスオフセットを求めた後、アドレス生成部1831はそれをx座標に加える。この和は、ソース画像から1ピクセルを求めるときに用いられる（ソースピクセルが補間されている場合は2ピクセル）。ソースピクセルが補間されている場合、アドレス生成部1831はx座標を次のインデックスオフセットに加え、2以上のピクセルを得る。

【0446】画像変換の座標を求めるとき、畳込み演算においても類似な手法を使う。畳込み演算との唯一の差異は、畳込み演算は次の出力ピクセルにおける行列の開始座標が前ピクセルにおける行列の開始座標から水平デルタだけ離れていることである。画像変換において、次のピクセルにおける行列の開始座標は、以前の出力ピクセルにおける行列の右上端ピクセルの座標から水平デルタだけ離れている。

【0447】図139において、中段のダイアグラムは上記の差を示す。プレ乗算部1832は必要であればピ

クセルのカラーチャンネルと不透明チャンネルを掛ける。補間部1832は必要なピクセルの真の色を求めるためソースピクセルを補間する。これはソース画像メモリから2ピクセルを取り、現在のx座標の分数部分を用いて補間し、その結果をレジスタに入力する。その後、ソース画像メモリの次の列の2ピクセルを取り、同じくxの分数を用いて補間する。その後、補間部1833は現在のy座標の分数部を用いて、この補間値とその前の補間値を補間する。

【0448】累算部1834は2つの作業をする。

1) 行列係数とピクセルを掛ける。
2) 全ての行列に対する上の結果を累算した値を次のステージに出力する。累算部1834の初期値は、チャンネルに応じて、0もしくは特定の値に初期化される。

【0449】ブロック1835は累算部1834の出力を切り捨て、必要であればアンダーフローやオーバーフローした値を最大値または最小値に制限する。そして、必要であれば出力の絶対値を求めることもある。累算部の出力において2進小数点の位置はカーネルディスクリプタのbpフィールドによって指定される。bpフィールドは、累算結果において捨てるべきビットの数を示す。これは、図139における下端のダイアグラムに示されている。この累算値は符号ありの2の補数として扱われる。

【0450】主データパス部242が行えるもう1つの画像処理動作は行列乗算である。行列乗算は2つの空間の間でアフィン関係がある場合の色空間変換に使われる。これが、(3次線形補間に基づく)一般色空間変換との差異である。行列乗算の結果は次の式によって定義される。

【0451】

【数7】

$$\begin{bmatrix} r_x \\ r_y \\ r_z \\ r_o \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ a_z \\ a_o \\ 255 \end{bmatrix}$$

【0452】ここで、 r_i は結果ピクセルであり、 a_i はAオペランドピクセルである。行列のサイズは5列4行でなければならない。図140は、主データパス部242において行列乗算を行なう乗算-加算器のブロック図である。この中にはピクセルチャンネルに行列係数を掛ける乗算部、その結果を合算する加算器、必要に応じて出力値をクランプしそして絶対値を求める論理部からなる。

【0453】行列乗算が終了するためには2クロックサイクルが必要である。各サイクルごとに多重化部を設定し、乗算部と加算部のデータが正しく選択されるようにする。第0サイクルにおいて、ピクセルの最下位2パイ

トが多重化部1851、1852によって選択される。次にその係数を行列の左側における2つの列、即ち、キャッシュにおける第0ラインにある行列係数に掛ける。

【0454】第1サイクルにおいて、ピクセルのより上位2バイトがトップ多重化部によって選択される。次にその係数を行列の右側における2つの列に掛ける。乗算の結果は最終サイクルの結果に加えられる1854。加算部における和は8ビットに切り捨てられる1855。

“オペランド論理部”1856は、加算部1854の入力が4つになるように乗算部出力を再配列する。これは乗算部の結果に対する加算を可能にするための再配列を行い、24ビット係数と8ビットピクセル成分との正しい積を出力するようにする。

【0455】“AC論理部”1855は加算部の出力の最下位12ビットを切り捨て、設定に従い切り捨てられた結果の絶対値を求める。その後、設定に応じて、その結果をクランプまたはラップする。“AC論理部”がクランプするように設定されたとき、0以下の全ての値は0に、255以上の全ての値は255に抑えられる。“AC論理部”がラップするように設定されたとき、定数部分の下位8ビットが出力される。

【0456】主データパス部242は、上記以外の画像処理を行なうように設定されることもできる。設計再利用によってコストが低減されるとともに、様々な画像処理動作を早く行なうことのできるコンピュータアーキテクチャについて以下述べるようにする。なお、このコンピュータアーキテクチャは柔軟性をもっているため、外部プログラミングエージェントであってもそのアーキテクチャにさえ慣れていれば、元々予測しなかった画像処理動作をも実行できるようにコンピュータを構成することができる。また、設計のコアは主にいくつかの多機能ブロックからなるため、設計の苦労を著しく減らすことができる。

【0457】3. 18. 6 データキャッシュ制御部とキャッシュ

データキャッシュ制御部240は、コプロセッサ224における4キロバイトの読み出しデータキャッシュ230を備えている。データキャッシュ230はダイレクトマップRAMキャッシュとして配列されており、外部メモリにおける同じ長さを持つラインのいずれも、キャッシュメモリ230(図2)における同じ長さの同じラインに直接マッピングされることができる。キャッシュメモリにおけるこのラインを普通キャッシュラインと呼び、上記キャッシュメモリは、多数のこのようなキャッシュラインからなる。

【0458】データキャッシュ制御部240は2つのオペランドオーガナイザ247、248からのデータ要求をサービスする。まずデータがキャッシュ230に存在するかを確認する。そうでなければデータが外部メモリからフェッチされる。データキャッシュ制御部240に

はプログラムのできるアドレス生成部があり、データキャッシュ制御部240がいくつかの異なるアドレッシングモードで動作するのを可能にする。また、要求されたデータのアドレスがデータキャッシュ制御部240によって作られるようになる特殊アドレッシングモードもある。このモードでは8ワード(256ビット)までのデータをオペレーションオーガナイザ247、248に同時に送ることができる。

【0459】キャッシュRAMは8つの独立してアドレス可能なメモリバンクからなる(異なるラインアドレスによってアドレスされた)。各々のバンクからのデータが256ビットに単位付けられる一部の特殊アドレッシングモードに必要である。この配置は、お互いに異なるバンクから来たものであれば、8つの32ビット要求までを同時にサービスすることができる。

【0460】キャッシュは、詳細に後述する以下のモードにおいて動作する。必要であれば、すべてのキャッシュが自動的に入れ込まれるようにすることも可能である。

1. ノーマルモード
2. 単一出力一般色空間変換モード
3. 多出力一般色空間変換モード
4. JPEG符号化モード
5. 低速JPEG復号モード
6. 行列乗算モード
7. デスケーブルモード
8. 無効化モード

図141は、図2におけるデータキャッシュ制御部240のアドレス、データ、制御フローとデータキャッシュ230とを示す。

【0461】データキャッシュ230は、前述したダイレクトマップキャッシュを具備する。データキャッシュ制御部240は、各キャッシュラインにおけるタグエントリを有するタグメモリ1872を具備しており、タグエントリはキャッシュラインが現在マップされている外部メモリアドレスの最上位部を有する。また、現在のキャッシュラインが有効であるかどうかを示すライン有効状態メモリ1873も備える。全てのキャッシュラインの初期状態は無効である。

【0462】データキャッシュ制御部240は、オペランドオーガナイザC247(図2)とオペランドオーガナイザC248(図2)からのデータ要求をオペランドバスインターフェースを通じて同時にサービスできる。動作において、オペランドオーガナイザ247、248(図2)のどちらかの一方もしくは両方はインデックス1874を提供し、データ要求信号1876を出す。アドレス生成部1881はインデックス1874に対して1つもしくはそれ以上の完全な外部アドレス1877を生成する。キャッシュ制御部1878は、生成されたアドレス1877のタグアドレスに対するタグメモリ1872を検査するとともに、関連するキャッシュラインが

有効であるかどうかを調べるためにライン有効状態メモリ1873を検査することにより、要求されたデータがキャッシュ230に存在するかどうかを判断する。要求されたデータがキャッシュメモリ230に存在するとき、要求データ1880と共に、アクノレジメント

(応答)信号1879が関連するオペレーションオーガナイザ247、248に送られる。要求されたデータがキャッシュメモリ230に存在しないとき、入力バスインターフェース1871と入力インターフェーススイッチ252(図2)を通じて、要求されたデータ1870が外部メモリからフェッチされる。データ1870は要求信号1882を出力し、要求されたデータ1870が生成されたアドレス1877を提供することによってフェッチされる。アクノリッジ信号1883及び要求されたデータ1870はそれぞれキャッシュ制御部1878及びキャッシュメモリ230に送られる。それから、そのキャッシュメモリ230に関連するキャッシュラインが新しいデータ1870によって更新される。新しいキャッシュラインのタグアドレスもタグメモリ1872に書き込まれ、新しいキャッシュラインにおけるライン有効状態1873が起動される。アクノリッジ信号1879はデータ1870とともに関連するオペランドオーガナイザ247又は248(図2)に送られる。

【0463】図142において、データキャッシュ230のメモリ構成を示す。データキャッシュ230は、キャッシュライン長が32である128個のキャッシュラインC0,...,C127をもつダイレクトマップキャッシュとして整理される。キャッシュRAMは別々のアドレス指定のできるメモリバンクB0,...,B7を具備しており、各メモリバンクは32ビットのバンクライン128個のを持ち、各キャッシュラインCiは8つのメモリバンクB0,...,B7において相当する8つのバンクラインB0i,...,B7iを有する。

【0464】生成された外部メモリアドレスの構成を図143に示す。生成されたアドレスは20ビットタグアドレス、7ビットラインアドレス、3ビットバンクアドレス、2ビットバイトアドレスからなる32ビットのワードである。20ビットタグアドレスはタグアドレスとタグメモリ1872に記憶されているタグと比較するのに使われる。7ビットラインアドレスはキャッシュメモリ1870にある関連するキャッシュラインのアドレスに使われる。3ビットバンクアドレスはキャッシュメモリ1870のメ関連するメモリバンクのアドレスに使われる。2ビットバイトアドレスは32ビットバンクラインの関連するバイトのアドレスに使われる。

【0465】図144は、データキャッシュ制御部240とデータキャッシュ230の構造のブロック図を示す。ここで、128×256ビットRAMはキャッシュメモリ230を構成し、これは8つの128×32ビットの分離住所付けが可能なメモリバンクからなる。この

RAMは書き込み可能ポート (write)、書き込みアドレスポート (write_addr)、書き込みデータポート (write_data) を持つ。また、読み可能ポート (read)、8つの読みアドレスポート (read_addr)、8つの読みデータ出力ポート (read_data) を持つ。キャッシュメモリ230の全てのメモリバンクへの同時書き込みを可能にさせるためキャッシュ制御ブロック1878から書き込み可能信号が生成される。必要によって、データキャッシュ230は書き込みデータポート (write_data) を通じて外部メモリからの1もしくはそれ以上のラインのデータに更新される。書き込みアドレスポート (write_addr) にラインアドレスを提供し、8:1多重化器MUXを利用することによって1ラインのデータが書き込まれる。8:1多重化器MUXはデータキャッシュ制御部 (addr_select) の制御の下で生成された外部アドレスからラインアドレスを選択する。キャッシュメモリ230の全てのメモリバンクへの同時読み込みを可能にさせるため、キャッシュ制御ブロック1878から読み可能信号が生成される。この方法で、キャッシュメモリ230のメモリバンクの8つの書きアドレスポート (read_addr) に提供される各々のラインアドレスに応じて、8つの読みデータポート (read_data) から8つの異なるバンクラインのデータを同時に読み込むことができる。

【0466】各々のキャッシュメモリ230のバンクはプログラム可能アドレス生成器1881を持っている。これは違う8つの位置への、関連する8つのメモリバンクからの同時アクセスを可能にする。各々のアドレス生成器1881はアドレス生成器1881の作動モード設定のためのdccモード入力、インデックスパケット入力、ベースアドレス入力、アドレス出力を持つ。プログラム可能アドレス生成器1881の作動モードは、

(a) dccモード入力への信号が各々のアドレス生成器1881をランダムアクセスモードにし、外部メモリアドレスがインデックスパケット入力へ提供され、一つもしくはそれ以上のアドレス生成器1881のアドレス出力に出力されるランダムアクセスモード;

(b) dccモード入力への信号が各々のアドレス生成器1881を適切なモードにするJPEGエンコーディングと復号、色空間変換、行列乗算モード。このモードでは、各々のアドレス生成器1881にはインデックスパケット入力へのインデックスが入力され、インデックスアドレスを生成する。作動モードによって、アドレス生成部は最大8つの異なる外部メモリアドレスを生成させることができる。

【0467】8つのアドレス生成部1881は8つの異なる論理回路からなっており、各々は入力としてベースアドレス、出力として外部メモリアドレスを持つdccモードとインデックスからなる。ベースアドレスレジス

タ1885はインデックスパケットの組合せである現在のベースアドレスを記憶し、dccモードレジスタ1888はデータキャッシュ制御部240の現在の作動モード(dccモード)を記憶する。

【0468】タグメモリ1872は1ブロック、128×20ビットのマルチポートRAMで構成される。このRAMは1つの書きポート(update_line_addr)、1つの書き可能ポート(write)、8つの読みポート(tag0_data, ..., tag7_data)を持っている。これは、8つのアドレス生成器1881が現在記憶されている、1つもしくはそれ以上に生成されたメモリアドレスの、ラインのタグアドレスを決定することによりポート(read0_line_addr, ..., read7_line_addr)において8つの同時のルックアップを可能にする。これらラインの現在のタグアドレスはポート(tag0_data, ..., tag7_data)からタグ比較部1886に出力される。ポート(update_line_addr)のタグメモリ1872への書き込みを可能にするため、必要によって、キャッシュ制御ブロック1872によりタグ書き信号は生成される。

【0469】128ビットのラインvalidメモリ1873は、キャッシュメモリ230の各キャッシュラインのvalid状態を保っている。これは1つの書きポート(update_line_addr)、1つの書き可能ポート(update)、8つの読み込みポート(read0_line_addr, ..., read7_line_addr)、8つの読み可能ポート(linevalid0, ..., linevalid7)からなる128×1ビットのメモリである。タグメモリと同じように、これは8つのアドレス生成部1881に、1つ若しくはそれ以上に生成されたメモリアドレスの個々のラインアドレスに対して、現在のラインにセーブされているラインvalid状態を決定させることにより、ポート(read0_line_addr, ..., read7_line_addr)に対しての8つの同時ルックアップを可能にする。このラインの現ラインvalidビットはポート(linevalid0, ..., linevalid7)からタグ比較部1886に出力される。必要によっては、ラインvalid状態メモリ1873の書きポートに、ポート(update_line_addr)からラインvalid状態メモリ1873への書き込みを可能にするための書き信号がキャッシュ制御ブロック1878から生成する。

【0470】タグ比較部1886は8つのタグ比較器からなっており、現在生成された外部アドレスのラインアドレスによってアクセスされるラインのタグメモリ1872に現在セーブされているタグアドレスを受け取るためのtag_data入力、現在生成された外部メモリアドレスのタグアドレスを受け取るためのtag_addr

r入力、比較されるタグアドレス部を設定するための現動作モード信号(`dcc_mode`)を受け取るための`dcc_input`、現在生成された外部アドレスのラインアドレスによってアクセスされるラインにあるライン`valid`状態メモリ1873に現在セーブされているライン`valid`状態を受け取るための`line_valid`入力を持っている。比較部1886は8つのアドレス生成部1881それぞれに対して8つの`hit`出力を持つ。生成された外部メモリアドレスのタグアドレスと、生成された外部メモリのラインアドレスによってアクセスされる位置にあるタグメモリ1872の内容とが一致する時、`hit`信号とそのラインへのライン`valid`状態ビット1873が出力される。この実施例では、外部メモリにセーブされているデータ構造は小さくなり、タグアドレスの最上位ビットが全て同じである。従って、タグアドレスの変化する最下位ビットだけを比較すれば良い。これはタグ比較部1886がタグアドレスの変化する最下位ビットを比較するよう現動作モード信号(`dcc_mode`)を設定することで可能になる。

【0471】キャッシュ制御部1878はキャッシュメモリ230にあるデータへのアクセスが可能なとき、オペランドB247、オペランドC248からの要求(`proc_req`)と通知(`proc_ack`)を受け取る。動作モードによっては、キャッシュメモリ230の8つまでのバンクから異なるアドレスのデータが要求される。要求データがキャッシュメモリ230からアクセスできる時、タグ比較部1886からそのメモリのラインにヒットを出す。出されたヒット信号(`hit0, ..., hit7`)に対して、キャッシュ制御部1878はポート(`cache_read`)に読み込み可能信号を生成し、ヒット信号が出されたキャッシュラインへの読み込みを可能にする。ヒット信号(`hit0, ..., hit7`)ではなく要求(`proc_req`)1876が出された時には、生成された要求(`ext_req`)と共にデータのキャッシュラインの外部メモリアドレスが外部メモリに送られる。このキャッシュラインは入力(`ext_data`)が可能な時、それを通じてキャッシュメモリ230の8つのバンクに書き込まれる。この場合、タグ情報もラインアドレスのタグメモリ1886に書き込まれ、そのラインのライン状態ビット1873が出力される。

【0472】キャッシュメモリ230の8つのバンクからのデータは、データオーガナイザ1892にあるいくつかの多重化器を通じて出力され、所定の方法で出力データパケット1894に位置付けられる。ある動作モードでデータオーガナイザ1892は、現動作モード信号(`dcc_mode`)と生成された外部メモリアドレスのバイトアドレス(`byte_addr`)を用いる事によって、8つのメモリバンクから出力された8つの32

ビットワードから8ビットワードを選択、出力することができる。他のモードでデータオーガナイザ1892は、8つのメモリバンクから出力された8つの32ビットワードを直接出力する。前述した通り、データオーガナイザはこのデータを決められた方式に整列し出力する。

【0473】要求は次の段階で行われる。

1) プロセッシングユニットはキャッシュ制御部1878にあるプロセッシングユニットインターフェースにアドレスを送りパケットデータを要求する。
2) 8つのアドレス生成ユニット1881は動作モードに従い、キャッシュメモリの各ブロックのアドレスを生成する。

【0474】3) 生成されたアドレスのタグ位置は3ポートのタグメモリ1886の4ブロックにセーブされているタグアドレスと比較され、8つの生成されたアドレスに相当するライン部によって位置づけられる。

4) それらが一致し、そのラインのライン`valid`状態1873が出されたら、要求されたデータはキャッシュメモリ230に存在するとみなされる。

【0475】5) 存在しないデータは外部バス1890を介してフェッチされ、キャッシュメモリ230の8つのブロックはその外部メモリからのデータラインの内容に更新される。新しいデータのタグアドレスはタグメモリ1886に書き込まれ、そのラインのライン`valid`状態1873が出される。

6) 全ての要求データがキャッシュメモリ230に存在すれば、それは決められたパケット形式でプロセッシングユニットに現れる。

【0476】前述した通り、コプロセッサ224の全ての部分(図2)は標準CBusインターフェース303(図20)を含めている。データキャッシュ制御部240とキャッシュ230の標準CBusインターフェースレジスタの詳細は、付録BのB42からB46までに記載されている。このレジスタの設定はデータ制御部240の作動を制御する。簡単のため、2つのレジスタ(`base_address`と`bcc_mode`)だけを図153に示す。

【0477】データキャッシュ制御部240とデータキャッシュ230が有効ならば、データキャッシュ制御部は最初全てのキャッシュラインを無効にして標準モードで動作する。ある命令の終わりには、データキャッシュ制御部240とキャッシュ230はいつも標準動作モードに切り替わる。”`Invalidate`”モードを除いた全てのモードには”`Auto-fill and validate`”と言うオプションがある。`dcc_cfg2`レジスタに1ビットをセットすることにより、全てのキャッシュを`base_address`レジスタにセーブされているアドレスから始めることができる。この動作の間、オペランドオーガナイザB、C247、

248からのデータ要求は中止される。キャッシュはこの動作が終わった後に有効になる。

a. 標準キャッシュモード

このモードでは、2つのオペランドオーガナイザにより要求データの外部メモリアドレスが提供される。アドレス生成部1881が外部メモリアドレスを出力し、内部タグメモリを用いてそれがメモリキャッシュ230に存在するのかを確かめる。両方の要求データがキャッシュ230に存在しない場合、入力インターフェーススイッチ252からデータが要求される。持続的かつ同時的要求に構えてラウンド・ロビンスケジューリングが採用される。

【0478】同時的な要求に対し、1つのデータアイテムがキャッシュに存在すれば、それは要求したデータバスの後ろの32ビットに位置するようになる。他のデータは入力インターフェーススイッチを通じて外部に要求される。

b. シングル出力一般色空間変換モード

このモードでは、要求はオペランドオーガナイザ部Bから12ビットバイトのアドレス形式で出される。図60に示されている様に、要求データアイテムは8ビットカラー出力値である。12ビットアドレスはアドレス生成部1881のindex_packet入力に入力され、8つのアドレス生成部1881は図96に示される形式の32ビット外部メモリアドレスを生成する。この生成されたアドレスのバンク、ライン、バイトアドレスは表12と図61によって決められる。外部メモリアドレスは、8つの9ビットラインとバイトアドレスとして解釈され、それはRAMの8つのバンクのバイトを指すために使われる。キャッシュは補間のため主データバス242によりオペランドオーガナイザ部に、図60に示された前述の原理で戻されたバンクの8バイト値を求めるためにアクセスされる。全てのシングル出力一般カラー値テーブルはキャッシュメモリ230に収まるため、シングルカラー変換モードを適用する前にシングル出力カラー値テーブルをキャッシュメモリ230にロードするのが望ましい。

c. マルチ出力一般色空間変換モード

このモードでは、12ビットワードアドレスがオペランドオーガナイザ部B247から受けられる。要求データアイテムは図62を参照して前述した32ビットカラー出力値である。12ビットアドレスはアドレス生成部1881のindex_packet入力に入力され、8つのアドレス生成部1881は、図96に示される形式の8つの異なる32ビット外部メモリアドレスを作る。外部メモリアドレスのラインとタグアドレスは、表12と図63によって決定される。外部メモリアドレスは、図63を参照して前述したように、7ビットラインアドレスと2ビットタグアドレスに分けられる9ビットアドレスを有する8個の9ビットアドレスとして解釈され

る。タグアドレスが発見されなかった場合、入力インターフェーススイッチ252（図2）から適切なデータがロードされるまでキャッシュは停止する。データが利用可能な場合、出力データはオペランドオーガナイザ部に出力される。

d. JPEG符号化モード

このモードでは、JPEG符号化モードに必要なテーブルなどがキャッシュRAMのバンクにセーブされる。テーブルの記憶についてはJPEG符号化モード（表14、16）のところに述べられている。

e. 低速JPEG復号モード

このモードでは、データは表17に従って生成される。

f. 行列乗算モード

このモードでは、キャッシュは256バイトラインのデータにアクセスするために使われる。

g. Disabledモード

このモードでは、全ての要求は入力インターフェーススイッチ252にパスされる。

h. Invalidate（無効化）モード

このモードでは、ラインvalid状態ビットをクリアすることにより、全てのキャッシュの内容が無効にされる。

【0479】3. 18. 7 入力インターフェーススイッチ

図2で、入力インターフェーススイッチはピクセルオーガナイザ部246、データキャッシュ制御部240、命令制御部235からの要求データを調節する役割を果たす。またこれは外部インターフェース制御部238とローカルメモリ制御部236に必要なアドレスとデータを伝送する。

【0480】入力インターフェーススイッチ252はベースアドレス若しくはホストメモリマップにあるメモリオブジェクトのいずれかのレジスタにその設定を保存する。20個のアドレスビットが必要なため、これはページ境界に整列されるバーチャルアドレスである。ピクセルオーガナイザ部、データキャッシュ制御部、命令制御部からの要求に対して、入力インターフェーススイッチ252は、まずデータの開始アドレスの上位6ビットからコプロセッサのベースアドレスビットを減じる。この結果が負であるか、この結果の上位6ビットが0ではない場合はPCIバスが望ましい伝送先であることを意味する。

【0481】結果の上位6ビットが0である場合は、データマップがコプロセッサのメモリ位置を現すことを意味する。その後、入力インターフェーススイッチはコプロセッサの位置が正しいか否かを判別するため次の3ビットを検査する。コプロセッサの正当な位置は、
1) コプロセッサのベースアドレスからオフセット0x01000000から始まる一般インターフェースが占める16メガバイト。

【0482】2) コプロセッサのメモリオブジェクトのベースアドレスからオフセット0x02000000から始まるローカルメモリ制御部(LMC)が占める32メガバイト。不当なコプロセッサの位置を指す要求は、入力インターフェーススイッチによりエラーと見なされる。PCIバスはコプロセッサのメモリオブジェクトが占める領域以外のアドレスのデータソースとなる。入力インターフェーススイッチは要求データがPCIバスからのものなのか、それとも一般インターフェースからのものかをEICに知らせるためiソース信号を用いる。

【0483】アドレス復号処理の後、正当な要求は適切なIBusインターフェースに伝送される。EICとLMCはi-a-c-k信号が出された時、入力インターフェーススイッチにデータを伝送する。しかし入力インターフェーススイッチは入力されるワード数をカウントしないので、現在のデータ伝送がいつ終わるのかを、ピクセルオーガナイザ部により制御されるi-o-e信号、命令制御部、データキャッシュ制御部が監視しなければならない。

【0484】入力インターフェーススイッチ252はピクセルオーガナイザ部、データキャッシュ制御部、命令制御部の3つのモジュールを調節する。これらはデータを同時に要求することができるが、物理的な資源は2つしかないため、その要求は直に処理されない。入力インターフェーススイッチに使われる調節技術は優先権をベースにし、またプログラムも可能である。入力インターフェーススイッチの設定レジスタにある制御ビットは、命令制御部、データキャッシュ制御部、ピクセルオーガナイザ部の相対的優先権を指定する。優先権が低いモジュールからの要求は、その他の2つのモジュールからの同じ資源へのアクセス要求がない場合に受け入れられる。少なくとも2つの要求発行元に同じ優先順位が与えられると、要求が受け付けられる発行元を決定するためにラウンドロビン技術を用いる必要が生じる。

【0485】1つのソースに直ちにアクセスするのが不可能であるため、入力インターフェーススイッチは要求されたデータのアドレスとバースト長を記憶し、要求元から提供されたデータをプリフェッチするかどうかをみる必要がある。あるソースに対する処理の中で、IBus処理がない場合には優先権を決める調整処理が必要になる。

【0486】図145に命令インターフェーススイッチ252の詳細を示す。スイッチ252は標準CBusインターフェースとレジスタファイル860以外にアドレス復号器863と調節部864の間に2つのIBusトランシーバ661を持つ。アドレス復号器863はピクセルオーガナイザ部、データキャッシュ制御部、命令制御部から受けた要求に対するアドレス復号をする。アドレス復号器863は、アドレスが正当なのかを検査する他、必要によってアドレスを再マッピングする。調節部

864はどの要求をIBusトランシーバ661からIBusトランシーバ662に伝送するのかを決める。優先権はプログラム可能である。

【0487】IBusトランシーバ861、862は、マルチプレクシングとデマルチプレクシング機能と、他のインターフェースから入力インターフェーススイッチへの通信を可能にするためのトライステートのバッファリング機能を有している。

3. 18. 8 ローカルメモリ制御部

図2において、ローカルメモリ制御部236は、ローカルメモリの制御及びローカルメモリとコプロセッサ内のモジュールとの間におけるアクセス要求の処理の全てを担当する。ローカルメモリ制御部236は、結果オーガナイザ249からの書き込み要求と入力インターフェーススイッチ252からの読み出し要求に回答する。更に、周辺インターフェース制御部237と通常の一般CBus入力からの読み出しと書き込み要求に対しても回答する。ローカルメモリ制御部はプログラム可能なプライオリティシステムを用いており、更にスループットを最大化するためにFIFOバッファを採用している。

【0488】本発明においては、ファーストイン・ファーストアウト(FIFO)バッファの他に、メモリアレイからポートをデカップルするためにマルチポートバーストダイナミックメモリ制御部が用いられている。図146は、本発明の第1の実施例に従い、4ポートバーストダイナミックメモリ制御部のブロック図を示している。この回路には、メモリアレイ1910へのアクセスを必要とする2つの書き込みポート(A1944とB1946)と2つの読み出しポート(C1948とD1950)が含まれている。読み出しポート1948、1950のデータパスは別個のFIFO1936、1938經由でメモリアレイ1910から出てくるのに対し、2つの書き込みポートからのデータパスは別個のFIFO1920、1922を通り、多重化部1912經由でメモリアレイ1910に向かう。中央制御部1932は、ダイナミックメモリ1910へのインターフェースに必要な全てのコントロール信号を駆動すると共に全体のポートアクセスを調整する。リフレッシュカウンタ1934は、メモリアレイ1910のためにダイナミックメモリのリフレッシュサイクルの必要時期を決め、制御部1932と共にこれらを調整する。

【0489】好ましくは、メモリアレイ1910に対するデータの読み出しと書き込みは、書き込みポート1944、1946からFIFO1920、1922へ、或はFIFO1936、1938から読み出しポート1948、1950への転送の2倍のレートで行われる。この結果、書き込みと読み出しポート1944、1946、1948、1950を通してデータを転送するのに要する時間に対し、メモリアレイ1910からの転送、又はメモリアレイ1910への転送に要する時間(いか

なるメモリシステムのボトルネックである)を可能な限り短くするのである。

【0490】データは、書き込みポート1944、1946のいずれかを經由してメモリアレイ1910に書き込まれる。書き込みポート1944、1946に接続された回路は、初期値ゼロのFIFO1920、1922のみを認知する事になる。書き込みポート1944、1946を通してのデータ転送は、FIFO1920、1922が一杯になるか、又はバーストが終了するまでスムーズに進んでいく。データが最初にFIFO1920、1922に書き込まれると、制御部1932はDRAMへのアクセスのための他のポートとの仲裁を行う。アクセスが得られると、データは最高レートでFIFO1920、1922から読み出され、メモリアレイ1910に書き込まれる。DRAM1910へのバースト書き込みサイクルは、FIFO1920、1922にプリセットされた数のデータワードが貯えられた場合、又は書き込みポートからのバーストが終了した場合のみに開始される。いずれの場合においても、DRAM1910へのバーストは許可された時点から進み、FIFO1920、1922が空になるか、又はより高いプライオリティポートからのサイクル要求があるまで続く。いずれのイベントにおいてもデータは、FIFOが充満するか、又は現在のバーストが終了し、新たなバーストを開始するまで、書き込みポートからFIFO1920、1922へ邪魔されなく続けて書き込まれる。後者の場合、新しいバーストは、以前のバーストがFIFO1920、1922を空にしてDRAM1910に書き込まれるまでは進行されない。前者の場合には、最初のワードがFIFO1920、1922から読み出されてDRAM1910に書き込まれるや否やデータ転送が再開される。FIFO1920、1922からのデータ転送が最高レートであるため、書き込みポート1944、1946がストールするのは、制御部1932が他のポートからのサイクル要求で割り込みされた時のみ可能である。書き込みポート1944、1946からFIFO1920、1922へのデータ転送に対するいかなる割り込みも、できるだけ最小に維持するのが望ましい。

【0491】読み出しポート1948、1950は逆の順で動作する。読み出しポート1948、1950が読み出し要求を出すと、即刻、DRAMサイクルが要求される。この要求に対する許可が得られるとメモリアレイ1910が読まれ、対応するFIFO1936、1938にデータが書き込まれる。最初のデータワードがFIFO1936、1938に書き込まれるやいなや、読み出しポート1948、1950による読み出しが可能になる。このように最初のデータワードを得るには初期遅延が存在するが、その後の連続するデータワードの獲得にはおそらくそれ以上の遅延は出て来ないのである。DRAMの読み出しは、より高いプライオリティのDRA

M要求があるか、読み出しFIFO1936、1938が一杯になった場合、或は読み出しポート1948、1950がそれ以上データを要求しなくなったら終了する。一旦このようにして読み出しが終了すると、FIFO1936、1938へプリセットされているデータワードの数に余裕ができるまで再開されない。一旦読み出しポートがサイクルを終了すると、FIFO1936、1938に残っているいかなるデータも廃棄される。

【0492】常にDRAMコントロールが最小値を上回るようにするため、プリセットされている数のデータワードが全て転送されるまで(或は、対応するFIFO1920、1922が空になるか、読み出しFIFO1936、1938が一杯になるまで)バーストが割り込みされないようにDRAMアクセスへの再仲裁は制限される。全てのアクセスポート1944、1946、1948、1950はそれぞれに対応するバースト開始アドレスを持っており、これらはバーストの開始時にカウンタ1942にラッチされている。このカウンタはポートに対する取り引きのためのカレントアドレスを保持しており、例え転送が割り込みされても、いつでも正しいメモリアドレスで再開する事が可能である。現在アクティブなDRAMサイクルのアドレスのみが多重化部1940により選択され、行アドレスカウンタ1916と列アドレスカウンタ1918に送られる。アドレスの低次Nビットは列カウンタ1918に入力され、一方の上位アドレスビットは行カウンタ1916へ入力される。多重化部1914は、DRAMの行アドレスタイムの間には行カウンタ1916からメモリアレイ1910へ行アドレスを出力し、DRAMの列アドレスタイムの間には列カウンタ1918から列アドレスを送る。行アドレスカウンタ1916と列アドレスカウンタ1918は、いかなるバーストの開始時においてもメモリアレイDRAM1910へロードされる。これは、ポートサイクルの開始時と、割り込みされたバーストの継続時の両方に当てはまる事実である。列アドレスカウンタ1918は、それぞれのメモリへの転送が起きた後にインクリメントされ、行アドレスカウンタ1916は列アドレスカウンタ1918がゼロに変わるとインクリメントされる。後者の場合にはバーストが終了され、新たな行アドレスで再開されなければならない。

【0493】本実施例では、メモリアレイ1910は4×8ビットバイトラインを含んでおり、ワード当たり32ビットを構成すると仮定している。更に、それぞれの書き込みポート1944、1946に対応する4バイトの書き込みイネーブル信号のセット1950、1952があり、個別的にデータがメモリアレイ1910内のそれぞれの32ビットデータワードのそれぞれの8ビット部分に書き込まれるようにする。メモリアレイ1910に書き込まれるそれぞれのワード内のいかなるバイトにデータの書き込みに対するマスクを任意にかける事が可

能であるため、対応するFIFO1926、1928にそれぞれのデータワードと共に書き込みイネーブル情報を貯えておく必要がある。これらのFIFO1926、1928は書き込みFIFO1920、1922のコントロールに用いられるのと同じ信号でコントロールされるが、FIFO1920、1922へのデータの書き込みに必要とされる32ビットの代わりに4ビットのみが用いられる。同様に、多重化部1930は多重化部1912と同じようにコントロールされる。選択された書き込みイネーブルは、制御部1932へ入力され、制御部はこれらの情報を用い、多重化部1912によりメモリアレイ1910へ入力される書き込みデータと同期してメモリアレイ1910内のアドレスされたワードへの書き込みを選択的に可能又は不可能にする。

【0494】図146の構成は制御部1932の制御下で動作する。図147は、図146において制御部1932の動作の詳細を示す状態図である。パワーアップの後とリセットの完了時に、状態器は強制的にIDLE100状態になり、この状態ですべてのDRAMコントロール信号がインアクティブ(high)になり、多重化部1914は行アドレスをDRAMアレイ1910へ送る。リフレッシュまたはサイクル要求が検出されると、RASDEL11962状態へ遷移される。次のクロックエッジでサイクル要求とリフレッシュがなくなったら、状態器はIDLE1900状態に戻る。そうでないと、DRAM tRP(RASプリチャージタイミング制限)周期が満たされた時にRASON1966状態へ遷移され、この時、行アドレスストローブ信号RASはローレベルになる。tRCD(RASからCASへの遅延タイミング制限)が満たされた後、COL1968状態へ遷移され、DRAMアレイ1910へ入力するための列アドレスを選択するように多重化部1914がスイッチされる。次のクロックエッジでCASON1970状態へ遷移され、DRAM列アドレスストローブ(CAS)信号がアクティブローになる。一旦、tCAS(CASアクティブタイミング制限)が満たされたら、CASOFF1972状態へ遷移され、この状態でDRAM列アドレスストローブ(CAS)は再びインアクティブハイになる。ここで、更なるデータワードが転送されることになっていると共に、より高いプライオリティのサイクル要求や、リフレッシュが差し迫っていないか、或は再仲裁するには速すぎる場合、それから一旦tCP(CASプリチャージタイミング制限)周期が満たされたらCASON1970状態へ復帰し、DRAM列アドレスストローブ(CAS)は再びアクティブローになる。もし更なるデータワードの転送がない、或は再仲裁が発生し、より高いプライオリティのサイクル要求や、リフレッシュが差し迫っている場合、tRAS(RASアクティブタイミング制限)とtCP(CASプリチャージタイミング制限)が両方満たされたら、その代わりにRA

SOFF1974状態へ遷移される。この状態で、DRAM行アドレスストローブ(RAS)信号はインアクティブハイになる。次のクロックエッジで状態器はIDLE1860状態に復帰し、次のサイクル開始を準備する。

【0495】RASDEL2 1964状態でリフレッシュ要求が検出されると、一旦tRP(RASプリチャージタイミング制限)が満たされたら、RCASON1980状態へ遷移される。この状態でDRAM列アドレスストローブがアクティブローになり、RASリフレッシュサイクルの前にDRAM CASを開始する。次のクロックエッジで遷移はRRASON 1978へ行われ、DRAM行アドレスストローブ(RAS)はアクティブローになる。tCAS(CASアクティブタイミング制限)が満たされると遷移はRCASOFF 1976へ行われ、DRAM列アドレスストローブ(CAS)はインアクティブハイになる。一旦tRAS(RASアクティブタイミング制限)が満たされると遷移はRASOFF1974へ行われ、DRAM行アドレスストローブ(RAS)はインアクティブハイになり、有効的にリフレッシュサイクルを終了させる。状態器は通常のDRAMサイクルのために上記のような振る舞いを継続し、IDLE 1960状態へ遷移する。

【0496】図146のリフレッシュカウンタ1934は単純にカウンタであり、15マイクロ秒当たり一回の固定レート、或は特殊DRAM業者の要求により定まったレートでリフレッシュ要求信号を発生させる。リフレッシュ要求が発行されると、この要求は図147の状態器により認知されるまで発行状態を続ける。このアノレジメントは、状態器がRCASON1980状態に入った時に行われ、状態器がリフレッシュ要求の撤去を検出するまでその状態を続ける。

【0497】図148には、疑似コードフォームで図146の仲裁器1924の動作が示されている。ここでは、4つのサイクル要求発行者の中でどれにメモリアレイ1910へのアクセスを許可するかを決める方法と、アクセスへの公平さを保つためにサイクル要求者のプライオリティを修正するメカニズムを記述している。これらのコードに用いられたシンボルは図149に説明されている。

【0498】それぞれの要求発行者は、その要求のプライオリティを表す4ビットを持っている。上位の2ビットは一般の構成レジスタに設定されている構成値により全般的なプライオリティにプリセットされている。プライオリティの下位2ビットは仲裁者24により更新される2ビットカウンタに収められている。仲裁の勝者を決める際に、仲裁者1924は単にそれぞれの要求者の4ビットの値を比較し、最高値の要求者にアクセスを許可する。要求者にサイクルが許可されると、下位2ビットのプライオリティカウンタの値はゼロになり、同一の上

位2ビットのプライオリティ値と勝者より低い下位2ビットのプライオリティ値を持つ他の要求者の下位2ビットのプライオリティカウンタは全て1ずつインクリメントされる。この結果、今メモリアレイ1910へのアクセスを許可された要求者は同一の上位2ビットプライオリティ値を持つ要求者の間で最も低いプライオリティになる。上位2ビットのプライオリティ値が勝者とは違った値を持つ要求者の下位2ビットのプライオリティ値は影響されない。プライオリティの上位2ビットの値は要求者の全般的なプライオリティを決め、下位2ビットの値は同一の上位プライオリティの要求者の間で公平な仲裁スキームを実現している。このスキームを用いることにより、ハードウェアで結線された固定プライオリティ（それぞれの要求者の上位2ビットがユニーク）から部分的な入れ替えと、部分ハードウェア結線（全てではないが、一部の上位2ビットプライオリティが他のと異なる）、厳密に公平な入れ替え（全ての上位2ビットのプライオリティ値が同一）までのいろいろな仲裁スキームが実現できる。

【0499】図149は、それぞれの要求者に対するプライオリティビットの構造とそのビットの利用法を示している。ここでは、図148に用いられているシンボルの意味も定義されている。上記の実施例で各種のFIFO1920、1922、1938、それから1936は幅32ビット、深さ32ワードである。この深さは効率と消費される回路エリアの間の良い線での妥協を与えている。しかし、深さの値は、パフォーマンスの変化と共に特定のアプリケーションのニーズに合わせて変えられる。

【0500】また、ここに示されている4ポート構成は単に一つの実施例である。メモリアレイと読み出しまたは書き込みポートのいずれかとの間に単一のFIFOバッファを用意するだけでも効果は得られる。しかし、多数の読み出しと書き込みポートを用いると最高のスピード向上が得られることになる。

3.18.9 他モジュール

他モジュール239は、コプロセッサ224の動作、リセット同期、内部診断信号を必要に応じて外部ピンにまわすことによるエラーと割り込み信号のマルチプレクシング、CBusの内部と外部フォームとの間のインタフェーシングや内部と一般Bus信号の一般/外部Bus出力ピンへのマルチプレクシングなどのためのクロックの発生と選択を行う。勿論他モジュール239の動作は、用いられるASICテクノロジーによるクロッキングへの要求と具現詳細により異なる。

【0501】3.18.10 外部インターフェース制御部

次に記述される本発明の特徴は、仮想メモリを共有するコプロセッサを有するホストコンピュータで仮想メモリを提供するための方法と装置に関連している。本発明の

実施例は、コプロセッサがホストプロセッサと連動し仮想メモリモードで動作可能になるよう模索している。

【0502】特に、コプロセッサはホストプロセッサの仮想メモリモードで動作することが可能である。コプロセッサには、ホストプロセッサの仮想メモリテーブルを参照することができる仮想メモリ対物理メモリマッピングデバイスが含まれており、コプロセッサにより生成された命令アドレスをホストプロセッサのメモリ内の対応する物理アドレスにマッピングする。むしろ、仮想メモリ対物理メモリマッピングデバイスは、グラフィックイメージを生成するためにコンピュータグラフィックコプロセッサの一部を形成する。コプロセッサには、イメージに種々の複雑な動作を行える多数のモジュールが含まれる。マッピングデバイスはコプロセッサとホストプロセッサとの間の相互作用に関与するのである。

【0503】外部インターフェース制御部(EIC)238は、コプロセッサのPCI Busと一般Busへのインターフェースを提供する。更に外部インターフェース制御部は、コプロセッサの内部仮想アドレス空間とホストシステムの物理アドレス空間との間をつなぐメモリマネージメントも提供する。外部インターフェース制御部238は、入力インターフェーススイッチ252からの要求に応じてホストメモリからデータを読み出す時や、結果オーガナイザ249からの要求に応じてホストメモリにデータを書き込む時にPCI Bus上のマスタとして作動する。PCI Busへのアクセスは、“PCI Local Bus Specification, draft 2.1” PCI special interest group, 1994の標準に従って具現する。

【0504】外部インターフェース制御部238は、入力インターフェーススイッチ252と結果オーガナイザ249からのPCI取り引きのための同時要求を仲裁する。仲裁は構成可能であるのが望ましい。受け取った要求のタイプには、一度にホストコプロセッサの1行以下のキャッシュライン読み出しや、ホストの1行と2行の間のキャッシュラインの読み出しと、2行又はそれ以上のキャッシュラインの読み出しが含まれる。長さ無制限の書き込みも外部インターフェース制御部238により具現される。更に外部インターフェース制御部238は、随意にデータのプリフェッチングも行う。

【0505】外部インターフェース制御部238の構築には、全てのコプロセッサの内部モジュールのために仮想メモリからホストの物理メモリへのアドレスマッピングを提供するメモリマネージメントが含まれる。このマッピングは、アクセスを要求するモジュールに対し完全に透明である。外部インターフェース制御部238がホストメモリへのアクセス要求を受け取ると、メモリマネージメントユニットを初期化して、その要求されたアドレスを変換する。メモリマネージメントユニットがアドレスの

変換に失敗すると、場合によっては一つまたはそれ以上のPCI Busの取り引きがアドレスの変換を完了する結果になる。これは、メモリマネジメントユニット自身がPCI Busへ取り引きを要求するもう一つのソースになれることを意味する。入力インターフェーススイッチ252や結果オーガナイザ249から要求されたバーストが仮想ページの境界を越えると、外部インターフェース制御部238は自動的にメモリマネジメントユニットを作動し、全ての仮想アドレスのマッピングを正しくやり直す。

【0506】メモリマネジメントユニット (MMU)

(図150の915)は、16個のルックアサイドバッファ (TLB) が基本になっている。TLBは仮想対物理アドレスマッピングのキャッシュとして作動する。TLBでは次のような作業が可能である。

1) 比較: 仮想アドレスが与えられると、TLBは対応する物理アドレスかTLBミス信号 (アドレスにマッチする有効なエントリがない場合) のいずれかを返す。

【0507】2) 置換: TLBには、既存エントリや有効でないエントリの代わりに新しい仮想対物理マッピングが書き込まれる。

3) 無効化: 仮想アドレスが与えられた時、TLBのエントリにマッチするとマッチしたエントリを無効化する。

4) 全無効化: すべてのTLBエントリを無効化する。

【0508】5) 読み出し: TLBエントリの仮想や物理アドレスは、4ビットアドレスベースで読み出される。テストのみに用いられる。

6) 書き込み: TLBエントリの仮想や物理アドレスは、4ビットアドレスベースで書き込まれる。TLB内のエントリは図151に示すようなフォーマットになっている。それぞれの有効なエントリは、20ビットの仮想アドレス670、20ビットの物理アドレス671、それから対応する物理ページが書き込み可能かを表すフラグで構成される。エントリの許容ページサイズは4Kバイトである。MMU内のレジスタは、比較に用いられた10ビットまでのアドレスにマスクをかけるのに用いることができる。これによってTLBのページは4Mバイトまでサポートされる。マスクレジスタは1つのみであるため、すべてのTLBエントリは同サイズのページを参照する。

【0509】TLBには、“Least-Recently Used” (LRU) 置換アルゴリズムが用いられている。新しいエントリは最も長い時間が経過したエントリに上書きされる。なぜなら、それは最後に書き込まれたか、或は比較作業で一致したものだからである。これは無効なエントリがない場合のみに適用される。無効なエントリがある場合には、有効なエントリに上書きする前に無効なエントリに書き込まれる。

【0510】図152はTLB比較操作の流れを示す。

受け取られた仮想アドレス880は881~883の3つの部分に分けられる。下位12ビット881は常にページ内のオフセットの部分であるため、対応する物理アドレスビット885へダイレクトに送られる。次の10ビット882は、マスクビットにより設定された通り、ページサイズによってオフセットの部分か、ページ番号の部分かのいずれかである。マスクレジスタ887内のゼロの値は、ビットがページオフセットの部分であるためTLB比較に用いてはいけないことを示している。10アドレスビットは10マスクビットとロジカルに“ANDED” (論理積) され、TLBルックアップのために下位10ビットの仮想ページ番号889を与える。仮想アドレスの上位10ビット883は、仮想ページ番号889の上位10ビットとしてダイレクトに用いられる。

【0511】このように生成された20ビットの仮想ページ番号はTLBに送られる。これがエントリの1つと一致すると、TLBは対応する物理ページ番号872と一致した位置の番号を返す。物理アドレス873は、マスクレジスタ887を再び用いて物理ページ番号から生成される。物理ページ番号872の上位10ビットは物理アドレス873の上位10ビットとしてダイレクトに用いられる。物理アドレス872の次の10ビットは、物理ページ番号 (対応するマスクビットが1の場合) か仮想アドレス (マスクビットが0の場合) かのいずれかから875に選択される。物理アドレスの下位12ビット885は仮想アドレスからダイレクトに与えられる。

【0512】最後に、マッチに従いLRUバッファ876が更新され、マッチされたアドレスの使用を表す。TLBミスは、入力インターフェーススイッチ252や結果オーガナイザ249がTLB872に存在しない仮想アドレスへのアクセスを要求した時に発生する。この場合、MMUは要求されたアクセスの処理を進める前に、ホストメモリ203のページテーブルから要求された仮想対物理変換をフェッチし、それをTLBに書き込まなければならない。

【0513】ページテーブルはホストメインメモリのハッシュテーブルである。それぞれのページテーブルエントリは、図153に示すようなフォーマットの2つの32ビットワードで構成されている。2番目のワードは物理アドレスのための上位20ビットを構成し、下位12ビットは予約されている。対応する仮想アドレスの上位20ビットは最初のワードに与えられている。下位12ビットには有効 (V) ビットと書き込み可能 (W) または“リードオンリ”ビットが含まれており、残りの10ビットは予約されている。

【0514】ページテーブルエントリには、基本的にTLBエントリと同じ情報が含まれている。ページテーブルの余分のフラグは予約されている。ページテーブル自身は、通常メインメモリ203内の複数のページにわた

って分散され、一般に仮想空間と隣接していて物理空間とは接していない。MMUには、ソフトウェアにより設定された16のページテーブルポインタのセットが含まれており、それぞれはページテーブルの部分を含んでいる4Kバイトメモリ領域への20ビットポインタである。これは、コプロセッサ224が64Kバイトサイズのページテーブルをサポートし、8Kページマッピングを有することを意味している。4Kバイトページサイズのシステムにおいて、これは最大32Mバイトのマッピングされた仮想アドレス空間を意味する。むしろページテーブルポインタは、TLBに用いられるページサイズとは関係なく、常に4Kバイトのメモリ領域を参照することである。

【0515】TLBミス後のMMU操作は、次のように図154の690に示している。

1. TLBに存在しない仮想ページ番号891上のハッシュファンクション892を実行し、ページテーブルへ13ビットのインデックスを生成する。

2. ページテーブルインデックス894、896の上位4ビット894を用い、ページテーブルポインタ895を選択する。

【0516】3. 20ビットのページテーブルポインタ895とページテーブルインデックス896の下位9ビットを連結し、最下位3ビットに000を設定することにより（ページテーブルエントリはホストメモリ内の8バイトを占めるため）、要求されたページテーブルエントリの物理アドレス890を生成する。

4. ページテーブルエントリの物理アドレス898から始め、ホストメモリから8バイトを読み出す。

【0517】5. 8バイトのページテーブルエントリ900がPCIバスへ返されたとき、VALIDビットが1にセットされていれば仮想ページ番号はTLBミスを起こした元の仮想ページ番号と比較される。両者がマッチしないと、上記のプロセスを用いて次のページテーブルエントリがフェッチされる（物理アドレスは8バイトずつインクリメントされる）。この過程はマッチする仮想ページ番号のページテーブルエントリが見つかるまで、或は無効なページテーブルエントリに遭うまで続けられる。無効なページテーブルエントリに遭った場合には、ページフォルトエラーが出され処理は中止する。

【0518】6. マッチする仮想ページ番号を有するページテーブルエントリが見つかったと、置換操作によって完全なエントリがTLBに書き込まれる。新しいエントリはLRUバッファ876によってポイントされたTLB位置に置かれる。それからTLBの比較作業が再び行われ、順調に続いて、元の要求されたホストメモリアクセスの処理が可能になる。新しいエントリがTLBに書き込まれると、LRUバッファ876は更新される。

【0519】EIC238に具現されているハッシュファンクション892は、20ビットの仮想ページ番号

(vpn)に対し、次の方程式を用いる。

$$index = ((vpn >> S1) \text{ XOR } (vpn >> S2) \text{ XOR } (vpn >> S3)) \& 0 \times 1 \text{ f f f};$$

ここで、S1、S2、S3は独立的にプログラム可能なシフト量（正、又は負）で、それぞれ4つの値を取ることができる。

【0520】ページテーブルの線形探索が4Kバイトの境界を越えると、MMUは自動的に次のページテーブルポインタを選択し、正しい物理メモリ位置で探索を継続する。この作業には、ページテーブルの最後から最初へのラッピングが含まれる。ページテーブルは、探索が常に終了されるように常に少なくとも1つの無効(null)エントリを含んでいる。

【0521】ソフトウェアがホストメモリ内のページを置換するたびに、新しい仮想ページのためのページテーブルエントリを追加し、置換されたページに対応するエントリを削除しなければならない。また、古いページテーブルエントリはコプロセッサ224のTLBにキャッシュされてはいけな。これは、MMU内のTLB無効化サイクルを果たすことにより行われる。

【0522】無効化サイクルは無効化作業を引き起こすビットと共に無効化される仮想ページ番号をし、MMUへのレジスタ書き込みを通じて果たされる。このレジスタ書き込みは、ソフトウェアによって直接、或は命令デコーダにより割り込みされた命令を通じて果たされる。無効化作業は、提供された仮想ページ番号のためにTLB上で果たされる。TLBエントリにマッチすると、エントリは無効にマークされ、無効化された位置が次の置換作業で用いられるようにLRUテーブルが更新される。

【0523】未決定の無効化作業はいかなる未決定のTLB比較より高いプライオリティを持っている。無効化作業が完了すると、MMUは無効化ビットをクリアし、次の無効化処理が可能であることを知らせる。MMUが要求された仮想アドレスのための有効なページテーブルエントリを見つけられない場合、これをページフォルトという。MMUはエラー信号を出し、フォルトを起こした仮想アドレスをソフトウェアがアクセス可能なレジスタに保管する。MMUはアイドル状態に入り、エラーが解決されるまで待機する。割り込みがクリアされると、MMUは次の要求された取り引きから再び作業を始める。

【0524】読み出し専用とマークされた（書き込み可能とマークされていない）ページへの書き込み作業がなされた時にもページフォルトが出される。外部インターフェース制御部(EIC)238は、一般バスへアドレスされている入力インターフェーススイッチ252と結果オーガナイザ249からの取り引き要求に応じられる。それぞれの要求モジュールは現在の要求が一般バス用かあるいはPCIバス用かを表す。入力インターフェース

スイッチ252と結果オーガナイザ249とのコミュニケーションに共通バスを用いるのとは異なり、一般バス要求へのEIC操作はPCI要求への操作と完全に分かれている。更にEIC238は、一般バス空間にダイレクトにアドレスするCbus取り引きタイプにも応じられる。

【0525】図150は、外部インターフェース制御部238の構造を示している。IBus要求は多重化部910を通り、多重化部910は要求の目的地をもとにして(PCIまたは一般バス) 適当な内部モジュールへ要求を導く。一般バスへの要求は、RBusとCBusも持っている一般バス制御部911へ送られる。RBus上の一般バスとPCIバス要求は異なるコントロール信号を用いるため、このバスには多重化部が必要とされない。

【0526】PCIバスへ導かれたIBus要求はIBusドライバ(IBD)912によって扱われる。同様に、PCIへのRBus要求はRBusレシーバ(RBR)914によって処理される。IBD912とRBR914は仮想アドレスを、物理アドレスを返すメモリマネジメントユニット(MMU)915に送る。IBD、RBR、それからMMUは、それぞれPCIトランザクションを要求できて、これらはPCIマスタモード制御部(PMC)917によって生成され、コントロールされる。IBDとMMUはPCI読み出しのみを要求し、RBRはPCI書き込みのみを要求する。

【0527】別個のPCIターゲットモード制御部(PTC)918は、ターゲットとしてコプロセッサへアドレスされた全てのPCIトランザクションを処理する。これはCbusマスタモード信号を命令制御部へ送り、すべての他モジュールへのアクセスを可能にする。PTCは、返されたCbusデータをPMC経由でPCIバスへ送るため、PCIデータバスビンのコントロールは単一のソースから出される。

【0528】EICレジスタとモジュールメモリへアドレスされたCbusトランザクションは標準Cbusインターフェース7によって扱われる。全てのサブモジュールはコントロールレジスタからビットをもらい、ステータスレジスタにビットを返す。これらは標準Cbusインターフェース内部に位置している。PCIバストランザクションのためのパリティ生成とチェックは、PMCとPTCのコントロール下で作動するパリティ生成とチェック(PGC)モジュール921によって処理される。生成されたパリティは、パリティエラー信号と同様にPCIバスへ送られる。パリティチェックの結果は、エラーレポートのためにPTCのコンフィギュレーションレジスタにも送られる。

【0529】図155は、図150のIBusドライバ912の構造を示している。受け入れたIBusアドレスとコントロール信号はサイクルの始点でラッチされる

930。オアゲート931はサイクルの始まりを検出し、コントロールロジック932に開始信号を発生する。仮想ページ番号を形成するラッチ930の上位アドレスビットはカウンタ935にロードされる。仮想ページ番号は、936にラッチされた物理ページ番号を返すMMU915(図150)へ送られる。

【0530】物理ページ番号と下位仮想アドレスビットは、マスク937によって再結合され、PMC717

(図102)へのPCI要求のためのアドレス938を形成する。また、サイクルのためのバーストカウンタもカウンタ939にロードされる。プリフェッチ動作は異なるカウンタ941とアドレスラッチと比較回路943を用いる。PMCから返されたデータは、データがプリフェッチの一部か否かを表すマーカと共にFIFO944にロードされる。データがFIFO944の前の部分で使用可能になってくると、ラッチ945、946経由で読み出し、ロジックによりクロックアウトされる。読み出しロジック946はIBusアクノレジメント信号も生成する。

【0531】中央コントロールブロック932は、状態器を含め、全てのアドレスとデータ要素の順次処理、それからPMCへのインターフェースをコントロールする。仮想ページ番号カウンタ935は、IBusアドレスからのページ番号ビットで、IBusトランザクションの開始と共にロードされる。この20ビットカウンタの上位10ビットは常に受け入れるアドレスからくる。下位10ビットに対しては、それぞれのビットは対応するマスクビット937が1にセットされていれば受け入れるアドレスからロードされ、そうでないと、カウンタビットが1にセットされる。20ビットの値はMMUインターフェースへ送られる。

【0532】通常の動作で、仮想ページ番号は初期アドレス変換の後で用いられない。しかし、IBDがバーストのページ境界越えを検出した場合には、仮想ページカウンタがインクリメントされ、もう1つの変換が行われる。カウンタがロードされた時仮想ページ番号の一部でない下位ビットが1にセットされているため、20ビットの値への単純インクリメントは実際のページ番号フィールドのインクリメントをもたらす。インクリメントされた後、次のインクリメントのためにカウンタをセットアップするために、マスクビット937が再び用いられる。

【0533】物理アドレスは、変換後、MMUが有効な物理ページ番号を返すたびにラッチされる936。マスクビットは、返された物理ページ番号と元の仮想アドレスビットとを正しく結合するために用いられる。物理アドレスカウンタ938は物理アドレスラッチ936からロードされる。これはPMCからワードが返されるたびにインクリメントされる。インクリメントされるたびにカウンタはモニタされ、トランザクションがページ境界

を越えようとしているか否かを判断する。マスクビットは、カウンタのどのビットが比較に用いられるかを判断するのに使用される。カウンタがページ内に残っているワードの数が2つ以下であることを検出すると、コントロールロジック932に信号を出し、2つのデータ転送後現在のPCI要求を終了し、必要に応じて新たなアドレス変換を要求する。カウンタは新しいアドレス変換後に再びロードされ、PCI要求が再開する。

【0534】バーストカウンタ939は、トランザクションの始点でIBusバースト値と共にロードされる6ビットのダウンカウンタである。これはPMCからワードが返されるたびにデクリメントされる。カウンタの値が2つ以下になると、コントロールロジック932へ信号を出し、これで2つのデータ転送後、PCIトランザクションを終了することができる（プリフェッチングが可能でない限り）。

【0535】プリフェッチアドレスレジスタ943は、いかなるプリフェッチの最初のワードの物理アドレスと共にロードされる。続くIBusトランザクションが開始し、それからプリフェッチカウンタが少なくとも1つのワードが巧くプリフェッチされたことを示したら、トランザクションの最初の物理アドレスがプリフェッチアドレスの値と比較される。両者がマッチすると、プリフェッチデータはIBus引取りを満たすのに用いられ、最後にプリフェッチされたワードの後のアドレスでPCIトランザクション要求が開始する。

【0536】プリフェッチカウンタ941は4ビットのカウンタで、プリフェッチ動作中にPMCによってワードが返されるたびに最大入力FIFOの深さと同じカウントまでインクリメントされる。続くIBusトランザクションがプリフェッチアドレスとマッチすると、プリフェッチカウンタがアドレスカウンタに足され、それからバーストカウンタから引かれ、PCI要求が要求される位置で開始できるようになる。代わりに、IBusトランザクションがプリフェッチされたデータの一部だけを必要とすると、要求されたバーストの長さはプリフェッチカウントから引かれ、それからラッチされたプリフェッチアドレスに足され、残りのプリフェッチデータは更なる要求を満たすために保留される。

【0537】データFIFO944は、8ワード×33ビットの非同期フォールスルーFIFOである。PMCからのデータは、データがプリフェッチの一部であるか否かを表すビットと共にFIFOに書きこまれる。FIFOの先端からのデータは、使用可能になるや否やFIFOから読み出されIBusへ送られる。データ読み出し信号を生成するロジックはclkと同期して動作し、IBusアクノレジメント出力を発生する。トランザクションがプリフェッチされたデータを用いて満たされる場合に、コントロールロジックからの信号は、FIFOから読み出すプリフェッチされたデータの数の情報を

を読み出しロジックに与える。

【0538】図156は、図150のRBusレシーバ914の構造を示している。コントロールは2つの状態器950、951との間でスプリットされる。書き込み状態器951はRBusへのインターフェースをコントロールする。入力アドレス752はRBusバーストの始点でラッチされる。バーストのそれぞれのデータワードは、バイトイネーブルと共にFIFO754に書き込まれる。FIFO954が充満するようになると書き込みロジック951によってr-レディが取り消され、オーガナイザがそれ以上のワードを書き込まないようにする。

【0539】書き込みロジック951は、再同期開始信号を介してメイン状態器950にRBusバーストの開始を通知し、オーガナイザがそれ以上のワードを書き込まないようにする。仮想ページ番号を形成する上位アドレスビットはカウンタ957にロードされる。仮想ページ番号はMMUへ送られ、MMUからは物理ページ番号958が返される。物理ページ番号と仮想アドレスの下位ビットはマスクに従って再結合され、カウンタ960にロードされ、PMCへのPCI要求のためのアドレスを提供する。PCI要求のそれぞれのワードのためのデータとバイトイネーブルは、すべてのPMCMインターフェースコントロール信号も扱うメインコントロールロジック950によってFIFO954からクロックアウトされる。メイン状態器は、ビジー信号を介してアクティブであることを示し、それは書き込み状態器へ再同期して返される。

【0540】書き込み状態器951は、r-ファイナルを用いてRBusバーストの終了を検出する。するとFIFO954へのデータのロードを中止し、メイン状態器にRBusバーストが終了したことを通知する。メイン状態器はデータFIFOが空になるまでPCI要求を継続する。それからビジーを取り消し、書き込み状態器が次のRBusバーストを開始するようにする。

【0541】図150に再び戻り、メモリマネジメントユニット915は、IBusドライバ（IBD）912とRBusレシーバ（IBR）914のために仮想ページ番号から物理ページ番号への変換を担当する。図157に、メモリマネジメントユニットの詳細を示している。16エントリの変換ルックアサイドバッファ（TLB）970は、TLBアドレスロジック971から入力データを受け取って出力を送り返す。状態器が含まれているTLBコントロールロジック972は、RBRまたはIBDからTLBアドレスロジックにバッファされている要求を受け取る。要求を受け取ると、入力のソースとTLBによって行われる作業を選択する。有効なTLB作業は、比較、無効化、全無効化、書き込みと読み出しである。TLB入力アドレスのソースとしては、IBDとRBRインターフェース（比較作業用）、ページテ

ープルエントリバッファ974 (TLBミisserビス用) またはTLBアドレスロジック内のレジスタなどがある。TLBは、TLBコントロールロジックにそれぞれの作業のステータスを返す。成功した比較作業からの物理ページ番号はIBDとRBRへ送り返す。TLBは最も最近アクセスされた(LRU)位置の記録を保有し、これはTLBアドレスロジックにとっては書き込み作業用の位置として用いるのに有用である。

【0542】比較作業が失敗した場合、TLBコントロールロジック972はページテーブルアクセスコントロールロジック976にPCI要求を開始するよう信号を出す。ページテーブルアドレスゼネレータ977は、内部ページテーブルポインタレジスタを用い、仮想ページ番号をもとにPCIアドレスを生成する。PCI要求から返されたデータは、ページテーブルエントリバッファ974へラッチされる。要求される仮想アドレスにマッチするページテーブルエントリが見つかり、物理ページ番号がTLBアドレスロジック977へ送られ、その後ページテーブルアクセスコントロールロジック976はページテーブルアクセスが完了したことを通知する。それからTLBコントロールロジック972は、TLBに新たなエントリを書き込み、比較作業を再び開始する。

【0543】SCIへのレジスタ信号とSCIからのレジスタ信号は両方の方向に再同期される980。信号は全てのサブモジュールへ行き来する。モジュールメモリインターフェース981は、標準CBusインターフェースからTLBとページテーブルポインタメモリ要素へのアクセスをデコードする。TLBアクセスは読み出し専用で、データを得るためにTLBコントロールロジックを用いる。ページテーブルポインタは読み出し・書き込み両方可能で、モジュールメモリインターフェースによってダイレクトにアクセスされる。これらのバスには同期回路も含まれている。

【0544】3.18.11 周辺インターフェース制御部図158には、図2の周辺インターフェース制御部(PIC)の一例を詳細に示している。PIC237は、外部周辺デバイスへ、又はデバイスからデータを転送するいくつかのモードの1つで動作する。基本的なモードは、

1) ビデオ出力モード: このモードで、データは外部ビデオクロックとクロック・データイネーブルのコントロール下で、周辺へ転送される。PIC237は、出力データに対し必要とされるタイミングで出力クロックとクロックイネーブルサインを送る。

【0545】2) ビデオ入力モード: このモードで、データは外部ビデオクロックとクロック・データイネーブルのコントロール下で、周辺へ転送される。

3) セントロニクスモード: このモードは、IEEE1284標準に定義されている標準プロトコルに従い、

周辺へと周辺からデータを転送する。

PIC237は、必要に応じて、内部データソースや目的地から外部インターフェースのプロトコルを分離する。内部データソースは、出力データの単一ストリームにデータを書き込み、選択されているモードによって外部周辺機器へ転送される。同様に、外部周辺からの全てのデータは単一入力データストリームに書き込まれ、可能な内部データ目的地の1つに要求されたトランザクションを満たすのに用いられる。

【0546】可能な出力データのソースとしては、LMC236 (ABusを用いる)、RO249 (RBusを用いる)、それから一般CBusの3つが挙げられる。PIC237は、これらのデータソースからのトランザクションに一度に1つのみに応答する。1つのソースからのトランザクションは次のソースが考慮される前に完全に終了するのである。一般に、いつでも1つのみのデータソースしかアクティブにはならないのである。2つ以上のソースがアクティブになった場合にはCBus、ABus、RBusのプライオリティで順に処理される。

【0547】通常通り、モジュールはPICの内部レジスタが含まれている標準CBusインターフェース990のコントロール下で動作する。更に、CBusインターフェース992は、コプロセッサ224を介して周辺デバイスをアクセスし、コントロールすることができる。ABusインターフェース991もローカルメモリ制御部とのメモリ相互作用を処理することができる。結果オーガナイザ249に加え、ABusインターフェース991とCBusインターフェース992は両方ともバイトワイドFIFOが含まれている出力データバス993へデータを送る。出力データバスへのアクセスは、どのソースが出力ストリームに対してプライオリティまたは所有権を持っているかを常にチェックする仲裁者によってコントロールされる。出力データバスは、どちらがイネーブルになっているかによってビデを出力制御部994とセントロニクス制御部997とインターフェースする。それぞれのモジュール994、997は出力データバスの内部FIFOから一度に1バイトを読み出す。セントロニクス制御部997は、周辺デバイスをコントロールするために標準セントロニクスデータインターフェースを具現する。ビデオ出力制御部には、要求されるビデオ出力プロトコルに従い、出力パッドをコントロールするロジックが含まれている。同様に、ビデオ入力制御部998には、用いられているいかなるビデオ入力標準もコントロールするロジックが含まれている。ビデオ入力制御部998は入力データバスユニット999へ出力を出し、これは再びビデオ入力制御部998かセントロニクス制御部997かのいずれかによって一度に1バイトずつ非同期でFIFOに書き込まれるデータとバイトワイド入力FIFOを構成する。

【0548】データタイマ996には種々のカウンタが含まれており、出力データバス993と入力データバス999内のFIFOの現在状態をモニタするために用いられている。以上のことから、コプロセッサを用いると多重イメージまたは単一イメージの多重部分を同時に生成するために二重ストリームの命令を実行するのが可能に思われる。一次命令ストリームは現在ページの出力イメージを得るのに用いられ、一次命令ストリームがアイドルになっている間に次のページのレンダリングを始めるために二次命令ストリームを用いることができる。その結果、標準モードの動作で、現在ページのイメージはレンダリングされてからJPEGコーデ241を用いて圧縮される。イメージをプリントする必要がある時に、コプロセッサ241は二度JPEGコーデ241を用いてJPEGエンコードドイメージを解凍する。出力デバイスにからそれ以上のJPEGデコードドイメージの部分が必要とされないアイドルタイムの間に、次のページまたはバンドの構成のために命令を実行するのが可能である。一般にこのプロセスは、コプロセッサの動作オーバーラップにより、イメージを生成するレートを上げる。特に、コプロセッサ224を用いると、コプロセッサに付いたプリンタによってプリントが行われ、結果的にレンダリングスピードが上がるため、イメージプロセッシング作業のスピードアップの面でベネフィットが得られるのである。

【0549】上記好適な実施例は本発明の1つの実施形態であり、本発明の範囲を外れずに当業者にとって自明な修正ができることが、以上から明らかであろう。

【0550】付録A

コプロセッサマイクロプログラミング

この節では新しい命令の実行毎にコプロセッサ内で行われる動作について詳述する。命令実行の間にコプロセッサにより行われるすべてのセルフコンフィグレーションは内部のレジスタのリード/ライトにより実現されており、従って、コプロセッサは外部のCバスインターフェースあるいはホストによってPCIバスインターフェースを用いることで完全にマイクロプログラミング可能である。但し、ホストを用いるマイクロプログラミングの場合には一般的にホスト同期の問題から困難となることが予想される。本章は読者がコプロセッサについて以下の点で十分な知識を持っていることを前提している。

1. 実行モデル
2. 命令セットとコーディング
3. レジスタセット
4. 内部構造

A. 1 一般事項

A1. 1 コプロセッサのセットアップに関する一般事項

コントロール命令とローカルDMA命令以外のすべての命令については、コプロセッサで内のデータの流れは基

本的にピクセルオーガナイザの制御下におかれる。ピクセルオーガナイザは入力データストリームの先頭のフェッチ、データのカウンタ、及び最後のデータがフェッチされた時期の決定について責任を持っている。コプロセッサ内のその他のモジュールは基本的に、送られてきたデータに単に応答するだけである。

A1. 2 モジュールのコンフィグレーション順序
すべてのモジュールが命令毎にセットアップされるわけではない。いくつかのモジュールは命令デコーディング時に、全くコンフィグレーションされない。モジュールのコンフィグレーション順序は常にPO, DCC, OOB, OOC, MDP, JC, RO, PICの順である。

A1. 3 その他のレジスタの設定

命令が、あるレジスタ値の設定を含んで符号化された場合にはそのレジスタは次の順序に従うマイクロプログラミングにより設定される。

1. 設定されるべきレジスタを持つモジュールに、ほかにレジスタセットが存在しなければ、そのレジスタはほかのいかなるレジスタ設定よりも先に設定される。
2. 設定されるべきレジスタを持つモジュールに、ほかにもレジスタセットがあるときはそのレジスタはほかのレジスタの設定が終わった後に、そのモジュールの_c f g レジスタの直前に設定される。

A1. 4 整合性のない命令オペランドのコーディング
多くの命令は、オペランド及び結果のデータタイプが指定されているので、ほかのデータタイプが指定された場合には、無意味な結果を返す。各オペランドに対し、コプロセッサは次の手順で目的のオペランドのフォーマットを決定する。

1. オペランドの内部フォーマットが1つのピクセル（圧縮バイトあるいは非圧縮バイト）に特化されている場合には、対応するオペランドオーガナイザはこれを反映して設定される。データキャッシュコントローラはコンフィグレーションされず、従ってノーマルモードで演算が継続される。
2. オペランドの内部フォーマットが「その他の形式」に特化されている場合には、コプロセッサは命令からオペランドのフォーマットを生成する。オペランドBとオペランドCについては前進的である。オペランドAについて「その他の形式」は元来指定されていなく、コプロセッサの振る舞いは定義されていない。対応するオペランドオーガナイザはバイパスモードになり、データキャッシュコントローラは得られたフォーマットのオペランドデータを管理するように設定される。マイクロプログラミングは合理的に様々なモジュール間で相互独立である。

A1. 5 疑似命令の文法

- ・命令の実行順序は左端の番号で決定される。
- ・レジスタ名はHelvetica Bold体でかかれている。

- ・レジスタフィールドは `register.field` によって示される。
- ・ `I`, `D` は現在復号化されている命令ワードとデータワードをそれぞれ示す。
- ・ `A`, `B` 及び `C` は現在復号化されているオペランドワード `A`、オペランドワード `B`、オペランドワード `C` を示す。
- ・ `A_descriptor`, `B_descriptor` および `C_descriptor` は現在復号化されている命令のデータワードのデスクリプタを示す。
- ・ `R` は現在復号化されている命令の結果ワードを示す。
- ・ `"X:Y"` は `X` と `Y` の連結を示す。
- ・ `@X` はコプロセッサのレジスタ番号 `X` を示す。
- ・ `"Cbus(X)"` は `C` バスオペレーション `X` の実行を示す。

- ・ `"*Cbus(X)"` は `C` バスオペレーション `X` による受け取りデータを示す。
- ・ `"*X"` は仮想メモリ番地 `X` を示す。
- ・ `"??"` は不明な値、あるいは未定の値を示す。
- ・ `"set"` はデータマニピュレーションレジスタの設定を示す。

A. 2 合成演算子

注:

1. 主要オペコードは `0xC` と `0xD`
2. 曖昧さは最上位アドレスのバイト (すなわち、最上位バイト) であると考える。
3. アキュムレータあるいはオペランドはプレ乗算されていてもよい。
4. 結果は非プレ乗算されていてもよい。
5. 命令長は入力ピクセルの数により定義されている。

-- Pixel Organiser (画素オーガナイザ):

```
if I.R=0 then
```

```
    po_len <- 0x0000:I.length
```

```
endif
```

```
if A_descriptor.S=0 then
```

```
    po_dmr <- set.
```

```
endif
```

```
po_said <- R
```

```
po_cfg.mode <- sequential    -- Set going
```

```
po_cfg.dst <- mdp
```

```
-- DCC in "normal32" mode (default)
```

--Operand Organiser B (オペランドオーガナイザB):

```
if B_descriptor.If != other then  --not blend
```

```
    oob_len <- po_len
```

```
    if B_descriptor.S = 0 then
```

```
        oob_dmr<- set
```

```
    endif
```

```
    oob_said <- A    --special for compositing
```

```
    if B_descriptor.what= tile
```

```
        oob_tile <- B
```

```
    endif
```



```

oob_cfg.operate <- operate

endif

-- Operand Organiser C (オペランドオーガナイザC) :

-- There is no implied other data type for this
.. bo used to specify any bit offset in a bit map_attenuation

if C_descriptor.if != other then

coc_len <- po_len

-- set up the ooc_dmr

if C_descriptor.S = 0 then

ooc_dmr <- set -- including bo

endif

ooc_said <- C

ooc_cfg.operate <- operate

endif

-- Main Data Path:

if B_descriptor.if = other then -- blend

mdp_len <- po_len

mdp_bm <- B

mdp_bi <- A

endif

mdp_cfg.instruction <- I.opcode

mdp_cfg.blendgen <- (B_descriptor.if = other)

mdp_cfg.decode < 1

mdp_cfg.operate <- 1

-- Result Organiser (結果オーガナイザ) :

if R_descriptor.S = 0 then

ro_dmr <- set

endif

ro_sa <- R

```

A. 3 ro_cfg.mode <- sequential
色空間変換

注:

1. 入力空間は常に3次元である。デフォルトでは3つの最下位なピクセルのチャンネルである。曖昧さは排除さ

れる。

2. カラーテーブルのフォーマットはひとつの出力チャンネルを含むものか、4つの出力チャンネルを含むもののうちどちらかである。

```

-- Pixel Organiser:
- Operand A only makes sense as source pixels, anything else
-- is probably wrong
if I.R = 0 then
    po_len <- 0x0000:I.len
endif

if A_descriptor.S = 0 then
    po_dmr <- set
endif

po_sald <- A
po_muv <- c      -- I & F tables
po_cfg-mode <- CSC
po_cfg.dst <- mdp

-- DCC setup:
-- should be other

if B_descriptor.if = other then
    dcc_addr <- B
    dcc_cfg2.cache_miss_inst <- B_descriptor.C
    if I.M= 0 then      --single output channel
        dcc_cfg2.mode <- single channel CSC
    else
        dcc_cfg2.mode <- multi channel CSC
    end
end

```

```

endif

endif

-- Operand Organiser B:

-- Operand B should be other, anything else is probably
-- wrong but do it anyway

if B_descriptor.if != other then

    oob_len <- po_len

    if B_descriptor.S= 0 then

        oob_dmr <- set

    endif

    oob_said <- B

    oob_cfg.operate <- operate

endif

-- Operand Organiser C:

-- Operand C should be other, anything else is probably
-- wrong but do it anyway

if C_descriptor.if != other then

    ooc_len <- po_len

    if C_descriptor.S= 0 then

        ooc_dmr <- set

    endif

    ooc_said <- C

    ooc_cfg.operate <- operate

endif

-- Main Data Path:

mdp_cfg.instruction <- I.opcode

mdp_cfg.decode <- 1

mdp_cfg.operate <- 1

```

-- Result Organiser:

if R_descriptor.S = 0 then

ro_dmr <- set

endif

ro_sa <- R

ro_cfg.mode <- sequential

A. 4 J P E G 命令

注:

1. オペコードは0x2である。
2. オペランドCはヤットするためのレジスタでもよい。

-- if there is a register set targeted at anything other than

-- the JC, PO, RO or DCC

if (D.R= 1) and (C_descriptor.topnibble !=0x5,0x6,0x9 or 0xB) then

@C_descriptor <- C

endif

-- Pixel Organiser:

-- Operand A is the source compressed data, and should be a

-- byte Stream

い。

3. オプションは多数存在する。

・サブサンプリングを行う／行わない。

・フィルタリングを行う／行わない。

・1, 3あるいは4スキャン。

4. これらの命令は命令実行前に設定されたいくつかのレジスタと関係している。

A. 4. 1 伸長

注: 1. 以下のレジスタは命令実行前に設定されている必要がある。

・ro_idr: 出力画像次元数レジスタ

・ro_cut: 出力カットレジスタ

・ro_lmt: 出力制限レジスタ

```

if I.R = 0 then
    po_len <- 0x0000:I.len
endif

if A_descriptor.S = 0 then
    po_dmr <- set
endif

po_said <- A

-- if there is a register set targeted at PO
if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
    @C_descriptor <- C
endif

po_cfg.mode <- sequential
po_cfg.dst <- jc

-- DCC setup:
if B_descriptor.if = other then
    dcc_addr <- B

    -- if there is a register set targeted at DCC
    if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
        @C_descriptor <- C
    endif

    dcc_cfg2.C <- B_descriptor.C
    dcc_cfg2.mode < JPEG decode
end if

-- Operand Organiser B:
-- Operand B should be other, anything else is probably
-- wrong, but do it anyway
if B_descriptor.if != other then
    oob_len <- po_len

    if B_descriptor.S = 0 then

```

```

        oob_dmr <- set
    endif

    oob_said <- B

    oob_cfg.operate <- operate
endif

-- Operand Organisr C:

-- Operand C may be a register to, set. If it isn't it should
-- be other, anything else is probably wrong, but do it anyway
if (R.D= 0) and (C_descriptor.if != other) then

    ooc_len <- po_len

    if C_descriptor.S= C then

        ooc_dmr <- set

    endif

    ooc_said <- C

    ooc_cfg.operate <- operate
endif

-- JC setup:

-- if there is a register set targeted at JC
if (D.R = 1) and (C_descriptor.topnibble=0x9) then

    @C_descriptor <- C

endif

jc_cfg.instruction <- I.opcode

jc_cfg.decode <- 1

jc_cfg.operate <- 1

-- Result Organisr:

if R_descriptor.S = 0 then

    ro_dmr <- set

endif

-- if there is a register write targeted at RO

```

```

ro_sa <- R
if (D.R = 1) and (C_descriptor.topnibble = 0xB) then
  @C_descriptor <- C
endif

ro_cfg.mode <- jpeg
ro_cfg.chan <- I.M:I.4
ro_cfg.upsample <- I.S
ro_cfg.cut <- I.C
ro_cfg.limit <- I.T

```

A. 4. 2 圧縮

注:

1. 以下のレジスタは命令実行前に設定されている必要がある。

- ・ po_idr : 出力画像次元数レジスタ
- ・ jc_rml : 再スタートマーカのインターバル
- ・ ro_cut : 出力カットレジスタ
- ・ ro_lmt : 出力制限レジスタ

```

-- if there is a register set targeted at anything other than
-- the JC, PO, .RO or DCC
if(D.R=1)and (C_descriptor.topnibble != 0x5,0x6,0x9 or 0xB) then
  @C_descriptor <- C
endif

-- Pixel Organiser:
-- Operand A is the source compressed data, and should be a
-- byte stream
if I.R = 0 then
  po_len <- 0x0000I.len
endif

if A_descriptor.S = 0 then
  po_dmr <- set

```

```

endif

po_said <- A

-- if there is a register set targeted at PO
if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
    @C_descriptor <- C
endif

po_cfg.mode <- jpeg
po_cfg.dst <- jc
po_cfg.ss <- I.S          -- subsampling
po_cfg.chan <- I.M:I4
po_cfg.F <- I,F

-- DCC setup:
if B_descriptor.if = Other then
    dcc_addr <- B
    -- if there is a register set targeted at DCC
    if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
        @C_descriptor <- C
    endif
    dcc_cfg2.cache_miss_inst <- B_descriptor.C
    dcc_cfg2.mode <- JPEG encode
end if

-- Operand Organiser B:
-- Operand B should be other ,anything else is probably
-- wrong, but do it anyway
if B_descriptor.if != other then
    oob_len <- po_len
    if B_descriptor.S = 0 then
        oob_dmr <- sel
    endif
endif

```



```

    oob_said <- B

    oob_cfg.operate <- operate
endif

-- Operand Organiser C:

-- Operand C may be a register to set. If it isn't it should
-- be other, anything else is probably Wrong, but do it
-- anyway

if (R.D = 0) and (C_descriptor.if != other) then

    ooc_len < po_len

    if C_descriptor.S= 0 then

        ooc_dmr <- set

    endif

    ooc_said <- C

    ooc_cfg.operate <- operate
endif

-- JPEG Coder:

-- if there is a register set targeted at JC

if (D.R=1) and (C_descriptor.topnibble = 0x9) then

    @C_descriptor <- C

endif

jc_cfg.instruction <- I.opcode

jc_cfg.decode <- 1

jc_cfg.operate <- 1

-- Result Organiser:

if R_descriptor.S = C then

    ro_dmr <- set

endif

ro_sa <- R

-- if there is a register write targeted at RO

```

```

if(D.R=1) and (C_descriptor.topnibble = 0xB) then

    @C_descriptor <- C

endif

ro_cfg.mode <- sequential

ro_cfg.cut <- I.C

ro_cfg.limit <- I.T

```

A. 5 データコーディング

注:

1. すべてのデータコーディング操作は圧縮、圧縮解除いずれの場合も同じ様に扱われる。これらの操作設定はJ P E Gの時とほとんど同じである。

2. 可能なエンコーディング操作

- ・ハフマン符号化
- ・予測符号化

3. 可能なデコーディング操作

- ・高速ハフマン復号化

・低速ハフマン復号化

・ p a c k b i t s 復号化 (バージョンA)

・ p a c k b i t s 復号化 (バージョンB)

・予測復号化

4. オペランドCは設定するためのレジスタでも良い。

5. 以下のレジスタは命令実行前に設定されている必要がある。

・ r o _ c u t : 出力カットレジスタ

・ r o _ l m t : 出力制限レジスタ

```

-- if there is a register set targeted at anything other than

-- the JC, PO, RO or DCC

if(D.R = 1) and (C_descriptor.topnibble != 0x5,0x6,0x9or0xB)then

    @C_descriptor <- C

endif

-- Pixel Organiser:

```

```

-- Operand A is the source data

if I.R = 0 then

    po_len <- 0x0000:I.len

endif

if A_descriptor.S = 0 then

    po_dmr <- set

endif

po_said <- A

-- if there is a register set targeted at PO

if (D.R = 1) and (C_descriptor.topnibble = 0x6) then

    @C_descriptor <- C

endif

po_cfg.mode <- sequential

po_cfg.dst <- jc

-- DCC setup:

if B_descriptor.if = other then

    dcc_addr <- B

-- if there is a register set targeted at DCC

    if (D.R = 1) and (C_descriptor.topnibble = 0x5) then

        @C_descriptor <- C

    endif

    dcc_cfg2.cache_miss_inst <- B_descriptor.C

    dcc_cfg2.mode <- en/de coding

else

-- if there is a register set targeted at DCC

    if (D.R = 1) and (C_descriptor.topnibble = 0x5) then

        @C_descriptor <- C

    endif

    dcc_cfg2.cache_miss_inst <- B_descriptor.C

```

```

endif

-- Operand Organiser B:

  Operand B Should be other ,anything else is probably
-- wrong, but do it anyway

if B_descriptor.if != other then

  oob_len <- po_len

  if B_descriptor.S= 0 then

    oob_dmr <- set

  endif

  oob_said <- B

  oob_cfg.operate <- operate

endif

-- Operand Organiser C:

-- Operand C may be a register to set. If it isn't it should
-- be other, anything else is probably wrong, but do it
-- anyway

if (R.D = 0) and (C_descriptor.if != other) then

  ooc_len <- po_len

  if C_descriptor.S= 0 then

    ooc_dmr <- set

  endif

  ooc_said <- C

  ooc_cfg.operate <- operate

endif

-- JPEG Coder:

--if there is a register set targeted at JC

if (D.R = 1) and (C_descriptor.topnibble = 0x9) then

  @C_descriptor <- C

endif

```

```

jc_cfg.instruction <- I.opcode

jc_cfg.decode <- 1

jc_cfg.operate <- 1

-- Result Organiser:

if R_descriptor.S = 0 then

    ro_dmr <- set

endif

ro_sa <- R

-- if there is a register write targeted at RO

if (D.R = 1) and (C_descriptor.topnibble = 0xB) then

    @C_descriptor <- C

endif

ro_cfg.mode <- sequential

ro_cfg.cut <- I.C

```

A. 6 変換と畳み込み $ro_cfg.limit <- I.T$

1. オペコードは 0×4 (畳み込み) と 0×5 (変換)。

2. コプロセッサは画像変換と画像畳み込みのそれぞれのために必要となるスーパーセットである操作を行う。画像変換と画像畳み込みの唯一の違いは、コプロセッサに関する限り、画像変換ではカーネルステップサイズがカーネルの大きさ (水平、垂直) なのに対して、畳み込みではステップサイズが 1 ソースピクセルとなっていること

である。

3. オプション:

- ・隣接ピクセルへのスナッピングおよび補間
- ・ピクセル (カーネル) の蓄積を行うか否か
- ・ソースピクセルのプレ乗算を行うか否か
- ・最終結果のクランプ、ラッピング、絶対値

4. 注: 変換と畳み込みは元の位置には実行できない。つまり、ソースのポインタとデスティネーションのポインタが同じであるときは、その内容が破壊される。

```

-- Pixel Organiser:

-- Operand A is the kernel descriptor and the PO delivers

-- Kernel co-efficients to the MDP. This is coded as

-- "Other"

    L=1 long format

--    0 short format

if I.R = 0 then

    po_len <- 0x0000:I.len

endif

if A_descriptor.S = 0 then

    po_dmr <- set

endif

po_said <- A

po_cfg.mode <- convolution/transformation

po_cfg.dst <- map

-- DCC setup:

-- the implied data type here is the source image

if B_descriptor.if = other then

    dcc_cfg2.cache_miss_inst <- B_descriptor.C

    if I.S = 0 then

        dcc_cfg2.mode <- 64 bit mode

    else

        dcc_cfg2.mode < random ,mode

    endif

else

    dcc_cfg2.cache_miss_inst <- B_descriptor.C

endif

-- Operand Organiser B:

    Operand B is a pointer to source image (other), anything

```

```

else is probably wrong, but do it anyway

if B_descriptor.if != other then

    oob_len <- po_len

    if B_descriptor.S = 0 then

        oob_dmr <- set

    endif

    oob_said <- B

    oob_cfg.operate <- operate

endif

-- Operand Organiser C:

-- Operand C descriptor is borrowed by the r. c field

-- so OOC is not set up

-- Main Data Path:

-- Kernel descriptor is delivered to MOP via PO

mdp_len <- po_len

mdp_bm.rows <- C_descriptor.r

mdp_bm.cols <- C_descriptor.c

mdp_cfg.instruction <- I.opcode

mdp_cfg.long_krnl <- A_descriptor.L

mdp_cfg.decode <- 1

mdp_cfg.operate <- 1

-- Result Organiser:

-- result will be pixels or part thereof

if R_descriptor.S = 0 then

    ro_dmr <- set

endif

ro_sa <- R

ro_cfg.mode <- sequential

```

A. 7 行列乗算

注:

1. オペコードは0 x 3
2. オプション:

- ・ソースピクセルのブレ乗算を行うか否か
- ・最終結果のクランプ、ラッピング、絶対値化
- ・オペランドCはレジスタに書き込んでも良い

```

- if there is a register set targeted at anything other than
-- the MDP, PO, RO or DCC

if(D.R=1) and(C_descriptor.topnibble != 0x5,0x6,0xA or 0xB) then

    @C_descriptor <- C
endif

-- Pixel Organiser:

-- Operand A is the source pixels. only makes sense to have
-- whole pixels, anything else is probably wrong

if I.R = 0 then

    po_len <- 0x0000:I.len
endif

if A_descriptor.S = 0 then

    po_dmr <- set
endif

po_said <- A

-- if there is a register Set targeted at PO

if(D.R=1)and (C_descriptor.topnibble = 0x6) then

    @C_descriptor <- C
endif

po_cfg.mode <- sequential

po_cfg.dst <- mdp

-- DCC setup:

-- the implied other data type is a matrix of coefficients

```



```

if B_descriptor.if = other then
    dcc_addr < B
endif

-- if there is a register write targeted at DCC
if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
    @C_descriptor <- C
endif

if B_descriptor.if = other
    dcc_cfg2.cache_miss_inst <- B_descriptor.C
    dcc_cfg2.mode <- matrix multiply
else
    dcc_cfg2.cache_miss_inst <- B_descriptor.C
endif

-- Operand Organiser B:
-- Operand B is the matrix co-efficients (other),
-- anything else is probably wrong, but do it anyway
if B_descriptor.if != other then
    oob_len <- po_len
    if B_descriptor.S = 0 then
        oob_dmr <- set
    endif
    oob_said <- B
    oob_cfg.operate <- operate
endif

-- Operand Organiser C:
-- Operand C may be a register to Set. If it isn't it should
-- be other, anything else is probably wrong but do it anyway
if (R.D = 0) and (C_descriptor.if != other) then
    ooc_len <- po_len

```

```

    if C_descriptor.S = 0 then
        ooc_dmr <- set
    endif

    ooc_said <- C

    ooc_cfg.operate <- operate
endif

-- MDP setup:
-- if there is a register Set targeted at MDP
if (D.R = 1) and (C_descriptor.ponibble = 0xA) then
    @C_descriptor <- C
endif

mdp_cfg.instruction <- I.opcode
mdp_cfg.decode <- 1
mdp_cfg.operate <- 1

-- Result Organiser:
if R_descriptor.S = 0 then
    ro_dmr <- Set
endif

ro_sa <- R

-- if there is a register write targeted at RO
if (D.R = 1) and (C_descriptor.topnibble = 0xB) then
    @C_descriptor <- C
endif

ro_cfg.mode <- sequential

```

A. 8 ハーフトーン処理

注:

1. オペコードは 0 x 7
2. オプションはハーフトーンのレベル値のみ

3. ハーフトーンスクリーンが適切にメッシュあるいはアンメッシュされているかぎり、ピクセルあるいはバイトに対して行うことができる。

```

-- Pixel Organiser:

-- A operand is the source pixels or bytes

--PO operates in conventional sequential mode

if I.R = 0 then

    po_len <- 0x0000:I.len
endif

if A_descriptor.S = 0 then

    po_dmr <- set
endif

po_said <- A

po_cfg.mode <- sequential

po_cfg.dst < mdp

-- DCC setup:

-- no implied other data type

-- if there is a register write targeted at DCC

if (D.R = 1) and (C_descriptor.topnibble = 0x5) then

    @C_descriptor <- C
endif

if B_descriptor.C = 0 then

    dcc_cfg2.cache_miss_inst <- B_descriptor.C
endif

-- Operand Organiser B:

-- Operand B is the halftone screen, and should be a tile

    anything else is probably wrong. There is no sensible

-- implied "other" data type for half toning.

if B_descriptor.if != other then

    obb_len <- po_len

    if B_descriptor.S= 0 then

```

```

        oob_dmr <- set
    endif

    oob_said <- B

    if B_descriptor.what= tile

        oob_tile <- C

    endif

    oob_cfg.operate <- operate

endif

-- Operand Organise C:

-- Operand C word is borrowed by the tiled Operand B.

-- The C_descriptor is completely ignored

-- OOC is in random mode (default) and wont be touched

-- Main Data Path:

mdp_bm.level <- I.levels

mdp_cfg.instruction <- I.opcode

mdp_cfg.decode <- 1

mdp_cfg.operate <- 1

-- Result Organiser:

if R_descriptor.S = 0 then

    ro_dmr <- set

endif

ro_sa <- R

ro_cfg.mode <- sequential

```

A. 9 メモリーコピー

注:

1. オペコードは0 x 9 2. この命令はメモリーコピーの操作を完了するために、全く個別の機構を用いている。

・汎用データ転送命令はコプロセッサにおける通常のデータフローを利用し、POおよびRO内のデータ操作ユ

ニットを用いる様々な関数を利用できる。

・ペリフェラルDMA命令はPICとLMC間の直接的なコネクションを利用する。このことはデータ操作ができないことを意味し、後続の命令と同時実行が可能である。

A. 9. 1 汎用データ転送

```

-- Pixel Organiser:

-- A Operand is the source data

-- PO operates in conventional sequential mode

if I.R = 0 then

    po_len <- 0x0000=I.len

endif

if A_descriptor.S = 0 then

    po_dmr <- B

end if

po_said <- A

po_cfg.mode <- sequential

po_cfg.dst <- jc

-- Operand Organiser B:

-- Operand B word represents a value to put in po_dmr

-- Operand Organiser C:

-- Operand C word represents the value to put in the ro_dmr

-- JPEG Coder:

jc_cfg.instruction <- I.opcode

jc_cfg.ibo <- D.bo

jc_cfg.obo <- I.obo

jc_cfg.decode <- 1

jc_cfg.operate <- 1

-- Result Organiser:

if R_descriptor.S = 0 then

    ro_dmr <- C

end

ro_sa <- R

ro_cfg.mode <- sequential

```

A. 9. 2 ペリフェラルDMA転送

注:

1. 同時実行でもそうでなくとも良い。このことは、I Cによって扱われている。
2. オペランドCは設定するレジスタでも良い
3. PICはデータを扱うモジュールなので、この命令はほかの”能動”命令と異なる。

```

--only setup required is for the PIC:

if I.S = 1 then      --data in from PIC

    pic_abus_addr <- R

else

    pic_abus_addr <- A

endif

-- if there is a register write targeted anywhere

if D.R = 1 then

    @C_descriptor <- C

endif

if I.R = 0 then

    pic_abus_cfg.ab_count <- 0x00:I.length

endif

pic_abus_cfg.ab_byte_en <- I.byte

pic_abus_cfg.ab_type <- I.R

pic_abus_cfg.start <- 1

-- Pixel Organiser:

-- Operand A is the source address given to the PIC

-- The PO is not setup

    -- Operand Organiser B:

    -- OOB is not setup

    --Operand Organiser C

    -- OOC is not setup

    -- Result Organiser:

    -- The Result word represents the target address

    -- RO is not set up

```

A. 10 フォトCD伸長

この命令群は3つの異なる操作すなわち、水平補間、垂直補間、残部融合から構成される。垂直補間と残部融合の設定方法は同じである。これら全ての命令のオペコードは0x9である。

A. 10. 1 水平補間

注:

1. ピクセルあるいはバイトに対して実行可能
2. この命令はオペランドが1つの命令であり、オペランドCは設定するレジスタでも良い。

```

-- if there is a register set targeted at anything other than
-- the MDP, PO or RO
if (D.R = 1) and (C_descriptor.topnibble != 0x6, 0xA or 0xB) then
    @C_descriptor <- C
endif

-- Pixel Organiser:
if I.R = 0 then
    po_len <- 0x0000:I.len
endif

if A_descriptor.S = 0 then
    po_drnr <- set
endif

po_said <- A

```

```

-- if there is a register set targeted at PO
if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
    @C_descriptor <- C
endif

po_cfg.mode <- sequential
po_cfg.dsl <- mdp

-- Operand Organiser B:
-- Operand a word is completely ignored
-- The B_descriptor is completely ignored
-- Operand Organiser C:
-- Operand C may be a register to Set up
-- Main Data Path:
-- if there is a register set targeted at MDP
if (D.R = 1) and (C_descriptor.topnibble = 0xA) then
    @C_descriptor <- C
endif

mdp_cfg.instruction <- I -opcode
if A_descriptor.if /= packed bytes then-- tell map about size
    mdp_cfg.word_input <- 1
else
    mdp_cfg.word_input <- 0
endif

mdp_cfg.decode <- 1
mdp_cfg.operate <- 1

-- Result Organiser:
if Rdescriptor.S = 0 then
    ro_dmr <- set
endif

ro_sa <- R

```

A. 1 0. 2. ro_cfg.mode <- sequential
 2. 垂直補間と残部融合

注:

1. 垂直補間と残部融合の設定は同じである。

2. ピクセルとバイトの両方に対して実行可能。

3. この命令はオペランドが2つの命令であり、オペランドCはレジスタセットでも良い。


```

-- if there is a register set targeted at anything other than
-- the MDP, PO, DOB or RO
if (D.R=1)and(C_descriptor.topnibble != 0x6,0x7,0xA or 0xB) then
    @C_descriptor <- C
endif

-- Pixel Organiser:
if I.R = 0 then
    po_len <- 0x0000:I.len
endif

if A_descriptor.S = 0 then
    po_dnr <- set
endif

po_said <- A

-- if there is a register set targeted at PO
if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
    @C_descriptor <- C
endif

po_cfg.mode <- sequential
po_cfg.dst <- mdp

-- DCC setup:
-- no implied other data type
-- if there is a register write targeted at DCC

```

```

if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
    @C_descriptor <- C
endif

if B_descriptor.C = C then
    dcc_cfg2.cache_miss_inst <- B_descriptor.C
endif

-- Operand Organiser B:
-- Operand B is the residuals or one operand for
-- interpolation, anything else is probably wrong, but
-- do it anyway
-- There is no sensible implied "other" data type for
-- this.
if B_descriptor.if != other then
    oob_len <- po_len
    if B_descriptor.S = 0 then
        oob_dmr <- set
    endif
    oob_said <- B
endif

-- if there is a register set targeted at OOB
if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
    @C_descriptor <- C
endif

cob_cfg.operate <- operate
endif

-- Operand Organiser C:
-- may be a register to set
-- Main Data Path:
-- if there is a register set targeted at MDP

```

```

if (D.R = 1) and (C_descriptor.topnibble = 0xA) then

    @C_descriptor <- C

endif

mdp_cfg.instruction <- I.opcode

if A_descriptor.if /= packed_bytes then-- tell mdp about size

    mdp_cfg.word_input <- 1

else

    mdp_cfg.word_input <- 0

endif

mdp_cfg.decode <- 1

mdp_cfg.operate <- 1

-- Result Organiser:

if R_descriptor.S = 0 then

    ro_dmr <- set

endif

ro_sa <- R

ro_cfg.mode <- 0          -- sequential

```

A. 11 制御命令

注:

1. 制御命令は2種類の操作、すなわちフロー制御命令と内部アクセス命令からなる。

A. 11.1 フロー制御

注:

1. オペコードは0 x B
2. フロー制御命令は現在、各種ジャンプ命令と各種の待機命令から成っている。
3. コプロセッサ内では明確な設置は行われず、またこの命令は、“能動”命令ではない。つまり、ほかの命令のようにコプロセッサ内のサブモジュールが実際に何かを行ったりはしない。
4. オペランドCは設定するレジスタでも良い。

-- only thing to do is:

```
if D.R = 1 then
```

```
    @C_descriptor <- C
```

```
endif
```

-- Pixel Organiser:

-- no setup associated with operand A

-- Operand Organiser B:

-- no setup associated with Operand B

-- Operand Organiser C:

-- no setup associated with Operand C

-- Result Organiser:

-- no setup associated with the Result

A. 11.2 内部アクセス (リード)

注:

1. オペコードは0 x A
2. リード命令はデータをコプロセッサ外に転送する。

3. ROが実際にコプロセッサ内ですべてを行う唯一のモジュールである。

```
-- Pixel Organiser:
-- no setup associated with Operand A
-- Operand Organiser B:
-- no setup associated with Operand B
-- Operand Organiser C:
-- no setup associated with Operand C
-- Result Organiser
ro_subst <- Cbus(A)
ro_sa <- B
ro_cfg.mode <- Cbus_data

Cbus(A) <- B-- Pixel Organiser:
-- no setup associated with Operand A
-- Operand Organiser B:
-- no setup associated with Operand B
-- Operand Organiser C:
-- no setup associated with Operand C
-- Result Organiser:
```

A. 1.2 予約された命令

注:

1. オペコード 0x0, 0xF は予約されている。
2. 予約された命令はマスク可能なエラーを出す。
3. これらの予約された命令はコプロセッサが今後改訂されたときにほかの命令として使用されることになっている。

付録B: レジスタ

1. 1 レジスタおよびテーブル

本節ではコプロセッサのレジスタについて解説する。これらのレジスタは3通りの方法で変更可能である。

1. 特定のコプロセッサの命令群はレジスタの読み書きをするためにある。これらの命令群を用いることでレジスタは、イニシエータのPICバスサイクルの開始あるいは汎用インターフェースのトランザクションを用いて、ローカルメモリインターフェースに関連するメモリへの、あるいはメモリからの読み書きが行われる。
2. 多くのレジスタは命令実行の副作用により内容が変化する。命令実行のためにコプロセッサが自身の設定を行うという主要な機構は、様々なレジスタを現在の状態

A. 1.1. 3 内部アクセス (ライト)

注:

1. オペコードは 0xA
2. ライト命令はデータをコプロセッサ内に転送する。
3. この命令は「能動」命令ではないので、IC以外のモジュールは実際には何も行わない。

を反映するように設定することで実現されている。命令実行終了後には各レジスタはコプロセッサの状態を反映する。多くの典型的な処理はある命令により完全に特定され、設定される。いくつかのレジスタでは命令実行の直前に設定する必要がある。

「予約」レジスタビットの意味

あらゆるレジスタ或はその構成要素の「予約」の意味は次の通りである。

・予約された場所への書き込みは行えるが、そのデータは棄却される。

・予約された場所からの読み込みは行えるが、そのデータは不定である

全ての特定されていないレジスタ及びレジスタフィールドは「予約」である。

1. 1. 1 レジスタの分類

コプロセッサ内のレジスタは本節に記述される振る舞いに基づいて分類される。これらの記述は

・外部: モジュール外部 (からのアクセス)。CBusインターフェースを用いた外部アクセスである。すなわち、命令コントローラあるいは外部CBusインターフ

エースによるターゲットモードのPCIを用いる。注、レジスタは、バイセットモードを介してPCIバスからセットできない。

・内部：モジュール内部（からのアクセス）

状態レジスタ

状態レジスタは外部からは読み込み専用で、内部からは読み書き可能。

コンフィグ1レジスタ

コンフィグ1レジスタは外部からは読み書き可能で、内部からは読み込み専用である。コンフィグ1レジスタはタイプCのCBus操作はサポートせず（すなわち、ビットセットモードをサポートしない）、アドレス値のようなバイト（またはそれより大きな）コンフィギュレーション情報を保持するレジスタとして用いられる。

コンフィグ2レジスタ

コンフィグ2レジスタも外部から読み書き可能で、内部からは読み込み専用である。コンフィグ2レジスタはタイプCのCBus操作（すなわちビットセットモード）をサポートし、ビット単位で設定する必要があるコンフィギュレーション情報を保持するレジスタとして用いられる。

コントロール1レジスタ

コントロール1レジスタは外部および内部から読み書き可能。コントロール1レジスタはタイプCのCBus操作をサポートせず（すなわちビットセットモードをサポ

ートしない）、アドレス値のようなバイト（またはそれより大きなコントロール情報を保持するレジスタとして用いられる。

コントロール2レジスタ

コントロール2レジスタは外部および内部から読み書き可能。コントロール2レジスタはタイプCのCBus操作（すなわちビットセットモード）をサポートし、ビット単位で設定する必要があるコントロール情報を保持するレジスタとして用いられる。

割り込みレジスタ

割り込みレジスタ内のビットは内部からは1にセットでき、外部からは1を書き込むことによって0にリセットできる。モジュール割り込み/エラーレジスタもこのタイプである。モジュールの割り込み/エラーレジスタは3つのフィールドから構成される。

〔7：0〕モジュールによって生成されたあらゆるエラー状態（ステータス）を意味する

〔23：8〕モジュールによって生成されたあらゆる例外状態を意味する

〔31：24〕モジュールによって生成されたあらゆる割り込み状態を意味する

1.1.2 レジスタマップ

表1.1はコプロセッサのレジスタである。番号はアドレスではなくレジスタ番号である。

表1.1 コプロセッサレジスタ

表 1.1 コプロセッサレジスタ

番号	名前	タイプ	説明	参照ページ
外部インターフェースコントローラレジスタ			26ページ	
0x00	eic_cfg	Config2	コンフィグレーション	
0x01	eic_stat	Status	ステータス	
0x02	eic_err_int	Interrupt	エラー及び割り込みステータス	
0x03	eic_err_int_en	Config2	エラー及び割り込みイネーブル	
0x04	eic_test	Config2	テストモード	
0x05	eic_gen_pob	Config2	汎用バスプロگرامブル出力ビット	
0x06	eic_high_addr	Config1	デュアルアドレスサイクルオフセット	
0x07				
0x08	eic_wtlb_v	Control2	仮想アドレス及びTLB無効化/書き込み用オペレーションビット	
0x09	eic_wtlb_p	Config2	TLB書き込み用物理アドレス及びコントロールビット	
0xDA	eic_mmu_v	Status	最新のトランスレータされたMMU仮想アドレス及び現在のLRU位置	
0xDB	eic_mmu_p	Status	MMUでフェッチされた最新のページテーブル物理アドレス	
0xDC	eic_ip_addr	Status	最新のPCIバスへのIバスアクセスの物理アドレス	
0xDD	eic_rp_addr	Status	最新のPCIバスへのRバスアクセスの物理アドレス	

表 1.1 コプロセッサレジスタ

番号	名前	タイプ	説明	参照ページ
0xDE	eic_iq_addr	Status	最新の汎用バスへのIバスアクセスの物理アドレス	

0x0F	eic_rg_addr	Status	最新の汎用バスへのIバスアクセスの物理アドレス	
ローカルメモリコントローラレジスタ				32ページ
0x10	lmi_cfg	Control2	汎用コンフィグレーションレジスタ	
0x11	lmi_sts	Status	汎用ステータスレジスタ	
0x12	lmi_err_int	Interrupt	エラー及び割り込みステータスレジスタ	
0x13	lmi_err_int_en	Control2	エラー及び割り込みイネーブルレジスタ	
0x14	lmi_dcfg	Control2	DRAMコンフィグレーションレジスタ	
0x15	lmi_mode	Control2	SDRAMモードレジスタ	
0x16				
0x17				
0x18				
0x19				
0x1A				
0x1B				
0x1C				
0x1D				
0x1E				
0x1F				

表 1.1 コプロセッサレジスタ

番号	名称	タイプ	説明	参照ページ
周辺インターフェースコントローラレジスタ				36ページ
0x20	pic_cfg	Config2	コンフィグレーション	
0x21	pic_stat	Status	ステータス	
0x22	pic_err_int	Interrupt	割り込み/エラーステータス	

		pt.		
0x23	pic_err_int_en	Config2	割り込み/エラーイネーブル	
0x24	pic_ahus_cfg	Control 2	コンフィグレーション及びAバスコントロール相	
0x25	pic_ahus_addr	Config1	Aバス転送用コプロセッサアドレス	
0x26	pic_cent_crg	Control 2	セントロニクスモード用コンフィグレーション及びコントロール	
0x27	pic_cent_dir	Config2	セントロニクスピン直接コントロールレジスタ	
0x28	pic_reverse_cfg	Control 2	逆(入力)データ変換のためのコンフィグレーション及び制御	
0x29				
0x2A	pic_timer0	Config1	初段データタイマ値	
0x2B	pic_timer1	Config1	後段データタイマ値	
0x2C				
0x2D				
0x2E				
0x2F				
他のモジュールレジスタ				ページ 8
0x30	mm_cfg	Config2	コンフィグレーションレジスタ	

表1.1 コプロセッサレジスタ

番号	名称	タイプ	説明	参照ページ
0x31	mm_stat	Status	ステータスレジスタ	
0x32	mm_err_int	Interrupt	エラー及び割り込みレジスタ	
0x33	mm_err_int_en	Config2	エラー及び割り込みマスク	

0x34	mm_gefg	Config2	グローバルコンフィグレーションレジスタ	
0x35	mm_diag	Config	自己診断コンフィグレーションレジスタ	
0x36	mm_greset	Config*	グローバルリセットレジスタ	
0x37	mm_gerr	Config2	グローバルエラーレジスタ	
0x38	mm_gexp	Config2	グローバルレジスタレジスタ	
0x39	mm_gint	Config2	グローバル割り込みレジスタ	
0x3A	mm_active	Status	グローバルアクティブ信号	
0x3B				
0x3C				
0x3D				
0x3E				
0x3F				
命令制御レジスタ				ページ 15
0x40	ic_cfg	Config2	コンフィグレーションレジスタ	
0x41	ic_stat	Status/ Interrupt	ステータスレジスタ	
0x42	ic_err_int	Interrupt	エラー及び割り込みレジスタ (エラークリア及び割り込みのための書き込み)	
0x43	ic_err_int_en	Config2	エラー及び割り込みレジスタ	

表1.1 コプロセッサレジスタ

番号	名称	タイプ	説明	参照ページ
0x44	ic_ipa	Control1	Aストリーム命令ポインタ	
0x45	ic_tda	Config1	TodoのAストリームレジスタ	
0x46	ic_fua	Control1	Aストリーム終了レジスタ	

0x47	ic_inta	Config1	Aストリーム書込みレジスタ	
0x48	ic_loa	Status	Aストリーム最終重畳命令列番号	
0x49	ic_ipb	Control1	Bストリーム命令ポインタ	
0x4A	ic_tdb	Config1	BストリームTodo レジスタ	
0x4B	ic_frb	Control1	Bストリーム終了レジスタ	
0x4C	ic_intb	Config 1	Bストリーム書込みレジスタ	
0x4D	ic_lcb	Status	Bストリーム最終重畳命令列番号	
0x4E	ic_sema	Status	Aストリームセマフォ	
0x4F	ic_semb	Status	Bストリームセマフォ	
データキャッシュコントロールレジスタ				ページ 42
0x50	dcc_cfB1	Config2	DCC コンフィグレーション1 レジスタ	
0x51	dcc_stat	status	ステートマシンステータスレジスタ	
0x52	dcc_err_int	status	DCC エラーステータスレジスタ	
0x53	dcc_err_int_en	Control1	DCC エラー書込みイネーブルビット	
0x54	dcc_cfg2	control2	DCC コンフィグレーション2 レジスタ	

表 1.1 コプロセッサレジスタ

番号	名称	タイプ	説明	参照ページ
0x55	dcc_addr	Config1	特別アドレスモード用ベースアドレス	
0x56	dcc_lv0	Control1	ライン0~31の“valid”ビットステータス	
0x57	dcc_lv1	Control1	ライン32~63の“valid”ビットステータス	
0x58	dcc_lv2	Control1	ライン64~95の“valid”ビットステータス	
0x59	dcc_lv3	Control1	ライン96~127の“valid”ビットステータス	

0x5A	dcc_raddrb	Status	オペランドオーガナイザB要求アドレス	
0x5B	dcc_raddrc	status	オペランドオーガナイザC要求アドレス	
0x5C	dcc_test	control1	DCC テストレジスタ	
0x5D				
0x5E				
0x5F				

画像オーガナイザレジスタ

ページ 50

0x60	po_cfg	Config2	コンフィグレーションレジスタ	
0x61	po_stat	Status	ステータスレジスタ	
0x62	po_err_int	Interrupt	エラー/書き込みステータスレジスタ	
0x63	po_err_int_en	Config2	エラー/書き込みイネーブルレジスタ	
0x64	po_dmr	Config2	データ操作レジスタ	
0x65	po_subst	Config2	代替値レジスタ	
0x66	po_odp	Status	現データポインタ	
0x67	po_len	Control1	長さレジスタ	
0x68	po_said	Control1	コプロセッサアドレスまたはデータ	
0x69	po_idr	Control 2	画像サイズレジスタ	
0x6A	po_mv_volid	Control2	MUV 有効ビット	
0x6B	po_mv	Config1	MUVRAMのベースアドレス	
0x6C				
0x6D				
0x6E				
0x6F				

オペランドオーガナイザBレジスタ

ページ 46

0x70	oob_cfg	Config2	コンフィグレーションレジスタ	
------	---------	---------	----------------	--

0x71	oob_stat	Status	ステータスレジスタ	
0x72	oob_err_int	Interrupt	エラー/割り込みレジスタ	
0x73	oob_err_int_en	Config2	エラー/割り込みイネーブルレジスタ	
0x74	oob_dmr	Config2	データ操作レジスタ	
0x75	oob_subst	Config2	代替値レジスタ	
0x76	oob_cdP	Status	現データレジスタ	
0x77	oob_len	Control1	入力長レジスタ	
0x78	oob_said	Control1	オペランドコプロセッサアドレス	
0x79	oob_tile	Control1	タイリング長/オフセットレジスタ	
0x7A				

表 1.1 コプロセッサレジスタ

番号	名称	タイプ	説明	参照ページ
0x7B				
0x7C				
0x7D				
0x7E				
0x7F				
オペランドオーガナイザCレジスタ				ページ16
0x80	ooc_cfg	Config2	コンフィグレーションレジスタ	
0x81	ooc_stat	Status	ステータスレジスタ	
0x82	ooc_err_int t	Interrupt	エラー/割り込みレジスタ	
0x83	ooc_err_int_en t_en	Config2	エラー/割り込みイネーブルレジスタ	
0x84	ooc_dmr	Config2	データ操作レジスタ	
0x85	ooc_subst	Config2	代替値レジスタ	

0x86	ooc_odp	Status	現データポインタ	
0x87	ooc_len	Control1	入力長レジスタ	
0x88	ooc_said	Control1	オペランドコプロセッサアドレス	
0x89	ooc_tile	Control1	タイリング長/オフセットレジスタ	
0x8A				
0x8B				
0x8C				
0x8D				
0x8E				
0x8F				
JPEG符号レジスタ				ページ63
0x90	jc_cfg	Config2	コンフィグレーション	
0x91	jc_stat	Status	ステータス	
0x92	jc_err_int	Interrupt	エラーと割込みステータスレジスタ	
0x93	jc_err_int _en	Config2	エラーと割込みイネーブルレジスタ	
0x94	jc_rsi	Config1	コプロセッサインターフェース	
0x95	jc_decode	Control2	現命令のデコード	
0x96	jc_res	Control1	残りの値	
0x97	jc_table_sel	Control2	復号した命令から表選択	
0x98				
0x99				
0x9A				
0x9B				
0x9C				
0x9D				

0x9E				
0x9F				
メインデータバスレジスタ				ページ54
0xA0	mdp_cfg	Config2	コンフィグレーション	

表 1.1 コプロセッサレジスタ

番号	名称	タイプ	説明	参照ページ
0xA1	mdp_stat	Status	ステータス	
0xA2	mdp_err_in t	Interruptu	エラー/割り込み	
0xA3	mdp_err_in t_en	Config2	エラー/割り込みイネーブル	
0xA4	mdp_test	Config2	テストモード	
0xA5	mdp_op1	Interrupt	現操作1	
0xA6	mdp_op2	Control2	現操作2	
0xA7	mdp_por	Control1	プラスオペレータのオフセット	
0xA8	mdp_bi	Control1	コプロセッサオフセットをインデック ステーブルエントリに混合	
0xA9	mdp_br	Control1	混合終了、又か序列の列又は行数 2 進数位置、階層のレベル数	
0xAA	mdp_len	Control1	混合長	
0xAB				
0xAC				
0xAD				
0xAE				
0xAF				
結果オーガナイザレジスタ				ページ68
0xB0	rc_cfg	Config2	コンフィグレーションレジスタ	

0xB1	ro_stat	Status	ステータスレジスタ	
0xB2	ro_err_int	Interrupt	エラー/割り込みレジスタ	
0xB3	ro_err_int_en	Config2	エラー/割り込みイネーブルレジスタ	
0xB4	ro_dat	Config2	データ操作レジスタ	
0xB5	ro_subst	Config1	代替値レジスタ	
0xB6	ro_cdP	Status	現データポインク	
0xB7	ro_len	Status	出力長レジスタ	
0xB8	ro_sa	Config 1	コプロセッサアドレス	
0xB9	ro_lcr	Config 1	画像サイズレジスタ	
0xBA				
0xBB	ro_vbase	Config1	コプロセッサ仮想ベースアドレス	
0xBC	ro_out	Config1	出力カットレジスタ	
0xBD	ro_lmt	Config1	出力長限界値	
0xBE				
0xBF				
PCIバスコンフィギュレーションスペースエイリアス				
0xCD-			PCI コンフィギュレーションスペース	ページ73
0xCD-			レジスタの0xC0から0xCと0xFの読み出し専用コピー	
0xCF-				
0xCE	pci_extrenal_cfg	Status	リセット時、外部シリアルROMからダウンロードされた32ビットフィールド。コプロセッサの動作は影響しない	

表 1.1 コプロセッサレジスタ

番号	名称	タイプ	説明	参照ページ
入出力インターフェーススイッチレジスタ				ページ23
0xD0	iis_cfg	Config2	コンフィギュレーションレジスタ	

0xD1	iis_stat	Status	ステータスレジスタ	
0xD2	iis_err_int	Interrupt	割込み/エラーステータスレジスタ	
0xD3	iis_err_int_en	Config2	割込み/エラーイネーブルレジスタ	
0xD4	iis_ic_addr	Status	ICからの入力アドレス	
0xD5	iis_dcc_addr	Status	DCCからの入力アドレス	
0xD6	iis_po_addr	Status	POからの入力アドレス	
0xD7	iis_burst	Status	PO, DCC及びICからのバースト長	
0xD8	iis_base_addr	Config1	ホストメモリマップにおけるコプロセッサメモリのベースアドレス	
0xD9	iis_test	Config1	テストモードレジスタ	
0xDA				
0xDB				
0xDC				
0xDD				
0xDE				
0xDF				
0xE0 - 0xFF			未使用	

1. 1. 3 レジスタ定義

汎用モジュールレジスタ

a. mm_cfg 表 1. 2 mm_cfg レジスタフィールド

フィールド	名称	説明
1	E	0=disabled (ディスエーブル) 1=enabled (イネーブル)
2	B	0=not bypass (バイパスしない) 1=bypassed (バイパス)

b. mm_stat 表 1.3 mm_stat レジスタフィールド

フィールド	名称	説明
		reserved

表 1.4 mm_err_int レジスタフィールド

フィールド	名称	説明
		reserved

表 1.5 mm_err_int_en レジスタフィールド

フィールド	名称	説明
		reserved

表 1.6 mm_gcfcg レジスタフィールド

フィールド	名称	説明
0	use_pci_clk	pci_clk信号を使用する値 (読取専用) 0=clk の生成に clk_in を使用 1=clk の生成に pci_clk を使用
1	slow_clk	ソースクロックを2で割って clk を生成 (読取専用) 0=2 (デフォルト) で割る 1=2で割らない

2	cbus_config	(読取り専用) 0=汎用バス構成 1=外部Cバス構成
31:1		Reserved

表 1.7 mm diag レジスタフィールド

フィールド	名称	説明
1:0	diag_cfg[1:0]	自己診断プログラムが diag_i ピンに表れる 00=コプロセッサがビジー 01=新たなAストリーム命令ストローブ 10=新たなBストリーム命令ストローブ 11=新たないずれかのストリーム命令ストローブ

3 : 2	diag_dcfg[1:0]	<p>自己診断プログラムが diag_d ピンに表れる</p> <p>00=自己診断アクティブ</p> <p>diag_d[0]=PCI バスマスタランザクション</p> <p>diag_d[1]=汎用バストランザクション</p> <p>diag_d[2]=ローカルメモリランザクション</p> <p>diag_d[3]=周辺インターフェースランザクション</p> <p>diag_d[4]=非重畳命令アクティブ</p> <p>diag_d[5]=重畳命令アクティブ</p> <p>01=命令自己診断</p> <p>diag_d[3:0] =現命令オペコード</p> <p>diag_d[4]=現命令ストリーム</p> <p>diag_d[5]=未使用</p> <p>10=キャッシング自己診断</p> <p>diag_d[0]=データキャッシュヒット</p> <p>diag_d[1]=データキャッシュミスのサービス</p> <p>diag_d[2]=TLB ヒット</p> <p>diag_d[3]=TLB ミスのサービス</p> <p>diag_d[4]=MUV ヒット</p> <p>diag_d[5]=MUV ミスのサービス</p> <p>11=自己診断ストール</p> <p>diag_d[0]=PO からの出力待ち</p> <p>diag_d[1]=ooB からの出力待ち</p> <p>diag_d[2]=00C からの出力待ち</p> <p>diag_d[3]=RO ファイルがフルのためのストール</p> <p>diag_d[4]=命令フェッチ待ち</p> <p>diag_d[5]=未使用</p>
-------	----------------	---

表 1.8 mm_grst レジスタフィールド

フィールド	名称	説明
15 : 0	msrst[15:0]	<p>モジュールソフトウェア毎のリセット 適当なビットに 1 を書込むことによりソフトウェアリセットこれらビットは自分でクリアされる</p> <p>bit 0=外部インターフェースコントローラ bit 1=ローカルメモリコントローラ bit 2=周辺インターフェースコントローラ bit 3=他のモジュール bit 4=命令コントローラ</p>

表 1.9 mm_gerr レジスタフィールド

フィールド	名称	説明
-------	----	----

31 : 16	err[15:0]	<p>モジュールごとのステータスビット</p> <p>これらビットは読出し専用。エラー状態をクリアするためにはモジュールエラービットをクリアしなければならない</p> <p>0=このモジュールでのエラーなし</p> <p>1=このモジュールでエラー</p> <p>bit 0=外部インターフェースコントローラ</p> <p>bit 1=ローカルメモリコントローラ</p> <p>bit 2=周辺インターフェースコントローラ</p> <p>bit 3=他のモジュール</p> <p>bit 4=命令コントローラ</p> <p>bit 5=データキャッシュコントローラ</p> <p>bit 6=画素オーガナイザ</p> <p>bit 7=オペランド・オーガナイザB</p> <p>bit 8=オペランド・オーガナイザC</p> <p>bit 9=JPEG符号化</p> <p>bit 10=メインデータパス</p> <p>bit 11=結果オーガナイザ</p> <p>bit 12=reserved</p> <p>bit 13=人カインターフェーススイッチ</p> <p>bit 14=reserved</p> <p>bit 15=どこかのモジュールでエラー</p>
---------	-----------	--

15 : 10	err_en[15:0 }	<p>モジュール毎のエラーイネーブル</p> <p>0=このモジュールからのエラー状態がホストの割り込みにならない 1=このモジュールからのエラー状態がホストの割り込みになる</p> <p>bit 0=外部インターフェースコントローラ</p> <p>bit 1=ローカルメモリコントローラ</p> <p>bit 2=周辺インターフェースコントローラ</p> <p>bit 3=他のモジュール</p> <p>bit 4=命令コントローラ</p> <p>bit 5=データキャッシュコントローラ</p> <p>bit 6=画素オーガナイザ</p> <p>bit 7=オペランド・オーガナイザB</p> <p>bit 8=オペランド・オーガナイザC</p> <p>bit 9=JPEG符号化</p> <p>bit 10=メインデータパス</p> <p>bit 11=結果オーガナイザ</p> <p>bit 12=reserved</p> <p>bit 13=入力インターフェーススイッチ</p> <p>bit 14=reserved</p> <p>bit 15=どこかのモジュールで割り込み</p>
---------	------------------	--

表 1.10 mm_gexp レジスタフィールド

フィールド	名称	説明
-------	----	----

31 : 16	exp[15:0]	<p>モジュール毎の例外ステータスビット</p> <p>これらビットは読出し専用。これら例外条件をクリアするためにはそのモジュールの例外ビットをクリアしなければならない</p> <p>0=このモジュールからの例外状態なし</p> <p>1=このモジュールから例外状態発生</p> <p>bit 0=外部インターフェースコントローラ</p> <p>bit 1=ローカルメモリコントローラ</p> <p>bit 2=周辺インターフェースコントローラ</p> <p>bit 3=他のモジュール</p> <p>bit 4=命令コントローラ</p> <p>bit 5=データキャッシュコントローラ</p> <p>bit 6=画素オーガナイザ</p> <p>bit 7=オペランド・オーガナイザB</p> <p>bit 8=オペランド・オーガナイザC</p> <p>bit 9=JPEG符号化</p> <p>bit 10=メインデータパス</p> <p>bit 11=結果オーガナイザ</p> <p>bit 12=reserved</p> <p>bit 13=入力インターフェーススイッチ</p> <p>bit 14=reserved</p> <p>bit 15=どこかのモジュールで例外</p>
---------	-----------	--

15 : 0	exp_en[15 : 0]	<p>モジュール毎の例外イネーブル</p> <p>0=このモジュールからの例外状態はホストの割込みにならない</p> <p>1=このモジュールから例外状態がホストの割込みになる</p> <p>bit 0=外部インターフェースコントローラ</p> <p>bit 1=ローカルメモリコントローラ</p> <p>bit 2=周辺インターフェースコントローラ</p> <p>bit 3=他のモジュール</p> <p>bit 4=命令コントローラ</p> <p>bit 5=データキャッシュコントローラ</p> <p>bit 6=画素オーガナイザ</p> <p>bit 7=オペランド・オーガナイザB</p> <p>bit 8=オペランド・オーガナイザC</p> <p>bit 9=JPEG符号化</p> <p>bit 10=メインデータパス</p> <p>bit 11=結果オーガナイザ</p> <p>bit 12=reserved</p> <p>bit 13=入力インターフェーススイッチ</p> <p>bit 14=reserved</p> <p>bit 15=どこかのモジュールで例外</p>
--------	----------------	---

表 1.11 mm_gint レジスタフィールド

フィールド	名称	説明
-------	----	----

31:16	int{15:0}	<p>モジュール毎の割込みステータスビット</p> <p>これラビとは読出し専用。割込み状態をクリアするためにはモジュールの割込みビットがクリアされなければならない</p> <p>0=このモジュールからの割込み状態なし</p> <p>1=このモジュールからの割込み状態</p> <p>bit 0=外部インターフェースコントローラ</p> <p>bit 1=ローカルメモリコントローラ</p> <p>bit 2=周辺インターフェースコントローラ</p> <p>bit 3=他のモジュール</p> <p>bit 4=命令コントローラ</p> <p>bit 5=データキャッシュコントローラ</p> <p>bit 6=画素オーガナイザ</p> <p>bit 7=オペランド・オーガナイザB</p> <p>bit 8=オペランド・オーガナイザC</p> <p>bit 9=JPEG符号化</p> <p>bit 10=メインデータパス</p> <p>bit 11=結果オーガナイザ</p> <p>bit 12=reserved</p> <p>bit 13=入力インターフェーススイッチ</p> <p>bit 14=reserved</p> <p>bit 15=どこかのモジュールで割込み</p>
-------	-----------	--

15 : 0	int_en[15 : 0]	<p>モジュール毎の割込みイネーブル</p> <p>0=このモジュールからの割込み状態がホストの割り込みにならない 1=このモジュールからの割込み状態がホストの割り込みになる</p> <p>bit 0=外部インターフェースコントローラ</p> <p>bit 1=ローカルメモリコントローラ</p> <p>bit 2=周辺インターフェースコントローラ</p> <p>bit 3=他のモジュール</p> <p>bit 4=命令コントローラ</p> <p>bit 5=データキャッシュコントローラ</p> <p>bit 6=画素オーガナイザ</p> <p>bit 7=オペランド・オーガナイザB</p> <p>bit 8=オペランド・オーガナイザC</p> <p>bit 9=JPEG符号化</p> <p>bit 10=メインデータパス</p> <p>bit 11=結果オーガナイザ</p> <p>bit 12=reserved</p> <p>bit 13=入力インターフェーススイッチ</p> <p>bit 14=reserved</p> <p>bit 15=どこかのモジュールで割込み</p>
--------	----------------	---

表 1.12 mm active レジスタフィールド

フィールド	名称	説明
-------	----	----

15:0	mactive[15:0]	<p>モジュール毎のアクティブ信号（読出し専用）</p> <p>これらビットはラッチされない</p> <p>bit 0=外部インターフェースコントローラ</p> <p>bit 1=ローカルメモリコントローラ</p> <p>bit 2=周辺インターフェースコントローラ</p> <p>bit 3=他のモジュール</p> <p>bit 4=命令コントローラ</p> <p>bit 5=データキャッシュコントローラ</p> <p>bit 6=画素オーガナイザ</p> <p>bit 7=オペランド・オーガナイザB</p> <p>bit 8=オペランド・オーガナイザC</p> <p>bit 9=JPEG符号化</p> <p>bit 10=メインデータパス</p> <p>bit 11=結果オーガナイザ</p> <p>bit 12=reserved</p> <p>bit 13=入力インターフェーススイッチ</p> <p>bit 14=reserved</p> <p>bit 15=どこかのモジュールがアクティブ</p>
------	---------------	--

命令コントローラレジスタ

1. ic_cfg

ic_cfg レジスタは3つの部分に別れる。最下位バイトはグローバルコンフィギュレーション情報を含む。

最下位から3番目のバイトはストリームAのコンフィギュレーション情報を含み、最上位バイトはストリームBのコンフィギュレーション情報を含む。このレジスタのリセット値は0x00000000である。

表 1.13 ic_cfg レジスタフィールド

フィールド	名称	説明
1	E	1=イネーブル 0=ディスエーブル

2	B	1=バイパスしない 0=バイパスする
3	S	0=自己テストモードでない 1=自己テストモード
[5:4]	pri[1:0]	命令ストリーム優先度 0=A (高い) , B (低い) 1=B (高い) , A (低い) 2=3=順番 (round robin)

フィールド	名称	説明
6	asr	非同期ストールリジェクト 0=イネーブルでない 1=イネーブル
7	pd	部分デコードイネーブル 0=イネーブルでない 1=イネーブル
8	cbus_eic_dis	Cbus サイクルの間 EIC からの要求ディス エーブル 0=ディスエーブルでない 1=ディスエーブル
9	cbus_mm_dis	Cbus サイクルの間 MM からの要求ディス エーブル 0=ディスエーブルでない 1=ディスエーブル
10	cbus_int_dis	Cbus サイクルの間 IC からの要求ディス エーブル 0=ディスエーブルでない 1=ディスエーブル
16	a_en	0=A ストリームディスエーブル 1=A ストリームイネーブル

17	a_seq	0=A ストリームシーケンス番号ディスエーブル 1= A ストリームシーケンス番号イネーブル
18	a_prefetch	0=A ストリーム先取りディスエーブル 1= A ストリーム先取りイネーブル
19	a_overlap_en	0=A ストリーム重畳ディスエーブル 1= A ストリーム重畳イネーブル
20	a_snc_pause	0=シーケンス番号完了割り込みで、A ストリームが停止しない 1= シーケンス番号完了割り込みで、A ストリームが停止
21	a_ic_pause	0=命令完了割り込みで、A ストリームが停止しない 1= 命令完了割り込みで、A ストリームが停止
22	a_aut_prime	0=A ストリームのシーケンス番号完了割り込みが自動的に起動しない 1= A ストリームのシーケンス番号完了割り込みが自動的に起動する
24	b_en	0=B ストリームディスエーブル 1=B ストリームイネーブル
25	b_seq	0=B ストリームシーケンス番号ディスエーブル 1= B ストリームシーケンス番号イネーブル
26	b_prefetch	0=B ストリーム先取りディスエーブル 1= B ストリーム先取りイネーブル

27	b_overlap_en	0=B ストリーム重畳ディスエーブル 1= B ストリーム重畳イネーブル
----	--------------	---

フィールド	名称	説明
28	b_snc_pause	0=シーケンス番号完了割り込みで、B ストリームが停止しない 1= シーケンス番号完了割り込みで、B ストリームが停止
29	b_ic_pause	0=命令完了割り込みで、B ストリームが停止しない 1= 命令完了割り込みで、B ストリームが停止
30	b_auto_prime	0=B ストリームのシーケンス番号完了割り込みが自動的に起動しない 1= B ストリームのシーケンス番号完了割り込みが自動的に起動する

m. is_stat

このレジスタは4つのセクションに分かれている。最下位バイトはICの内部状態を保持する。最下位から2番目のバイトは現在の命令の復号化された結果と現在及びプリフェッチした命令ストリームを保持する。最上位か

ら2番目のバイトはAストリームに関してすべてのステータス情報を保持する。最上位バイトはBストリームに関する情報を保持する。このレジスタのリセット値は0x00000000である。

表 1.14 ic_stat レジスタフィールド

フィールド	名称	説明
-------	----	----

[3:0]	ec_state[3:0]	命令実行状況 0=アイドル 1=命令フェッチ 2=重畳命令が終了するのを待つ 3=復号 4=命令先取り 5=命令の終了を待つ 6=レジスタの更新前に外部アクセスとの同期をとる 7=ステートレジスタの更新 8=悪い状態
8	overlap	0=重畳命令が実行中でない 1=重畳命令実行中
9	ic_interruption	0=完了時に割込みなし 1=完了時に割込み
10	jump	0=現命令がジャンプ命令でない 1=現命令がジャンプ命令
11	prefetched	0=次の命令が先取りバッファに先取りされていない 1=次の命令が先取りバッファに先取りされている

フィールド	名称	説明
12	stream	現実行中のストリーム（非重畳命令） 0=ストリーム A 1=ストリーム B

13	pref_stream	先取りした命令ストリーム 0=ストリーム A 1=ストリーム B
14	condition	0=条件が不一致の時にジャンプ 1=条件一致でジャンプ
16	a_wait	0=A ストリームが通常実行中 1=A ストリームがセマフォにより停止
17	a_async	0=A ストリームで非同期の転送が進行中でない 1=A ストリームで非同期の転送が進行中 1=Asynchronous transfer in progress in stream A
18	a_busy	0=A ストリームがビジーでない 1=A ストリームがビジー
19	a_sem	A ストリームレジスタのセマフォステータス： 0=セマフォが要求されていないか、ハードウェアにより要求されている 1=セマフォが外部で保持されている セマフォを受取った代行者はこのビットが0であることがわかる
20	a_lock	0=A ストリームがロックされていない 1=A ストリームがロックされている
21	a_primed	0=A ストリームの“シーケンス番号完了”割り込みが起動しない 1=A ストリームの“シーケンス番号完了”割り込みが起動

22	a_paused	A ストリーム停止 0=A ストリームがエラー割り込みで停止しない 1=A ストリームがエラー割り込みで停止。 コプロセッサが 1 を書込むことにより実行
23	a_ol_primed	0=A ストリームの“重畳命令シーケンス番号完了”割り込みが起動しない 1=A ストリームの“重畳命令シーケンス番号完了”割り込みが起動
24	b_wait	0=B ストリームが通常実行中 1=B ストリームがセマフォにより停止
25	b_async	0=B ストリームで非同期の転送が進行中でない 1=B ストリームで非同期の転送が進行中 1=Asynchronous transfer in progress in stream A
26	b_busy	0=B ストリームがビジーでない 1=B ストリームがビジー

フィールド	名称	説明
27	b_sem	B ストリームレジスタのセマフォステータス： 0=セマフォが要求されていないか、ハードウェアにより要求されている 1=セマフォが外部で保持されている セマフォを受取った代行者はこのビットが 0 であることがわかる

23	b_lock	0=Bストリームがロックされていない 1=Bストリームがロックされている
29	b_primed	0=Bストリームの“シーケンス番号完了” 割り込みが起動しない 1=Bストリームの“シーケンス番号完了” 割り込みが起動
30	b_paused	Bストリーム停止 0=Bストリームがエラー割り込みで停止 しない 1=Bストリームがエラー割り込みで停止 。コプロセッサが1を書込むことにより実行
31	b_ol_prime d	0=Bストリームの“重畳命令シーケンス 番号完了”割り込みが起動しない 1=Bストリームの“重畳命令シーケンス 番号完了”割り込みが起動

n. ic_err_int

このレジスタはIC内部で割り込みやエラーが発生した

かどうかを示す、アクティブ・ハイのフラグを含む。それぞれのビットは1を書き込むことでクリアされる。

表 1.15 ic_err_int レジスタフィールド

フィールド	名称	説明
8	a_ill_err	A ストリーム不正命令エラー
16	b_ill_err	B ストリーム不正命令エラー
24	a_snc_int	A ストリーム“シーケンス番号完了”割り込み
25	a_ic_int	A ストリーム“命令完了”割り込み

26	a_sns_int	A ストリーム“コプロセッサでされたシーケ ンス番号” 割込み
27	a_is_int	A ストリーム “コプロセッサでされた命令” 割込み
28	b_snc_int	Bストリーム “シーケンス番号完了” 割込み
29	b_ic_int	Bストリーム “命令完了” 割込み
30	b_sns_int	Bストリーム“コプロセッサでされたシーケ ンス番号” 割込み
31	b_is_int	Bストリーム “コプロセッサでされた命令” 割込み

o. ic_err_int_en を含み、リセット値は0x00000000である。
このレジスタは様々なエラーや割り込みの許可のマスク

表 1.16 ic_err_int_en レジスタフィールド

フィールド	名称	説明
8	a_ill_en	A ストリーム不正命令エラーイネーブル
16	b_ill_en	B ストリーム不正命令エラーイネーブル
24	a_snc_int_en	A ストリーム “シーケンス番号完了” 割込み イネーブル
25	a_ic_int_en	A ストリーム “命令完了” 割込みイネーブル
26	a_sns_int	A ストリーム“コプロセッサでされたシーケ ンス番号” 割込みイネーブル
27	a_is_int	A ストリーム “コプロセッサでされた命令” 割込みイネーブル
28	b_snc_int	Bストリーム “シーケンス番号完了” 割込み イネーブル
29	b_ic_int	Bストリーム “命令完了” 割込みイネーブル

30	b_sns_int	Bストリーム“コプロセッサでされたシーケ ンス番号” 割込みイネーブル
31	b_is_int	Bストリーム “コプロセッサでされた命令” 割込みイネーブル

p. ic_ipa 最下位ビットは命令が整列されてるはずであるとして0
このレジスタはストリームAの命令フェッチに用いられ
る仮想アドレスの最上位30ビットを保持する。2つの
に仮定される。このレジスタのリセット値は0x000
00000である。

表 1.17 ic_ipa レジスタフィールド

フィールド	名称	説明
[31:0]	ipa[31:2]	Aストリーム命令ポインタ

q. ic_tda

このレジスタはストリームAの“todo”値を保持する。これは適正な命令が存在するまでの32ビット

(ラッピング)のシーケンス番号である。このレジスタのリセット値は0x00000000である。

表 1.18 ic_tda レジスタフィールド

フィールド	名称	説明
[31:0]	tpa[31:0]	Aストリーム“todo”値

r. ic_fna

このレジスタはストリームAの「終了」値を保持する。これは32ビット (ラッピング) のシーケンス番号で最

後に完了した命令を示している。このレジスタのリセット値は0x00000000である。

表 1.19 ic_fna レジスタフィールド

フィールド	名称	説明
[31:0]	fna[31:0]	Aストリーム“完了”値

s. ic_inta

このレジスタはストリームAの「割り込み」番号を保持する。これは機構が有効であり用意されている場合にど

こへ割り込みをかけるかの、32ビット (ラッピング) のシーケンス番号である。このレジスタのリセット値は0x00000000である。

表 1.20 ic_inta レジスタフィールド

フィールド	名称	説明
[31:0]	inta[31:0]	Aストリーム“割り込み”値

t. ic_loa

このレジスタはストリームAで実行される最後の重複命令の32ビット (ラッピング) のシーケンス番号を保持

する。このレジスタのリセット値は0x00000000である。

表 1.21 ic_loa レジスタフィールド

フィールド	名称	説明
[31:0]	loa[31:0]	Aストリーム“最終重複命令シーケンス”値

u. ic_ipb

このレジスタはストリームBの命令フェッチに用いられる仮想アドレスの最上位30ビットを保持する。2つの

最下位ビットは命令が整列されているはずであるとして0に仮定される。このレジスタのリセット値は0x00000000である。

表 1.22 ic_ipb レジスタフィールド

フィールド	名称	説明
[31:0]	ipb[31:2]	Bストリーム命令ポインタ

v. ic_tdp

このレジスタはストリームBの“todo”値を保持する。これは適正な命令が存在するまでの32ビット

(ラッピング)番号である。このレジスタのリセット値は0x00000000である。

表 1.23 ic_tdp レジスタフィールド

フィールド	名称	説明
[31:0]	tdb[31:0]	Bストリーム"todo"値

w. ic_fnb

このレジスタはストリームBの「終了」値を保持する。
これは32ビット（ラッピング）のシーケンス番号で最

後に完了した命令を示している。このレジスタのリセット値は0x00000000である。

表 1.24 ic_fnb レジスタフィールド

フィールド	名称	説明
[31:0]	fnb[31:0]	Bストリーム“完了”値

x. ic_intb

このレジスタはストリームBの「割り込み」番号を保持する。これは機構が有効であり用意されている場合にど

こへ割り込みをかけるかの、32ビット（ラッピング）のシーケンス番号である。このレジスタのリセット値は0x00000000である。

表 1.25 ic_intb レジスタフィールド

フィールド	名称	説明
[31:0]	intb[31:0]	Bストリーム“割り込み”値

y. ic_lob

このレジスタはストリームBで実行される最後の重複命令の32ビット（ラッピング）のシーケンス番号を保持

する。このレジスタのリセット値は0x00000000である。

表 1.26 ic_lob レジスタフィールド

フィールド	名称	説明
[31:0]	lob[31:0]	Bストリーム“最終重畳命令シーケンス”値

z. ic_sema

このレジスタはic_statレジスタの副作用を用いたエイリアスであり、このレジスタの読み込みはストリームAのレジスタセマフォの要求の副作用である。

aa. ic_semb

このレジスタはic_statレジスタの副作用を用いたエイリアスであり、このレジスタの読み込みはストリームBのレジスタセマフォの要求の副作用である。

入力インターフェースレジスタ

ab. iis_cfg

表 1.27 iis_cfg レジスタフィールド

フィールド	名称	説明
[31:14]		Reserved
[31:12]	po_p	PO の優先度 (0=最低、1=最高)
[11:10]	dcc_p	DCC の優先度 (0=最低、1=最高)
[9:8]	ic_p	IC の優先度 (0=最低、1=最高)
[7:4]		Reserved
[3]	s	0=自己テストモードでない 1=自己テストモード
[2]	b	0=バイパスしない 1=バイパスする
[1]	e	0=ディスエーブル 1=イネーブル

a c. i i s _ s t a t

表 1.28 iis_stat レジスタフィールド

フィールド (ビット)	名称	説明
[31 : 25]		Reserved
[24]	i_prefetch_po	i_prefetch_po 信号のステート
[23]	i_prefetch_dcc	i_prefetch_dcc 信号のステート
[22]	i_prefetch_ic	i_prefetch_ic 信号のステート
[21 : 20]	lmc_po_lpri	IMC 用の PO 裁量の優先度の交替
[19 : 18]	lmc_dcc_lpri	IMC 用の DCC 裁量の優先度の交替
[17 : 16]	lmc_ic_lpri	IMC 用の IC 裁量の優先度の交替
[15 : 14]	eic_po_lpri	EIC 用の PO 裁量の優先度の交替
[13 : 12]	eic_dcc_lpri	EIC 用の DCC 裁量の優先度の交替
[11 : 10]	eic_ic_lpri	EIC 用の IC 裁量の優先度の交替
9	lmc_gnt_po	1=PO が IMC の制御許可
8	lmc_gnt_dcc	1=DCC が IMC の制御許可
7	lmc_gnt_ic	1=IC が IMC の制御許可
6	eic_gnt_po	1=PO が EIC の制御許可
5	eic_gnt_dcc	1=DCC が LEIC の制御許可
4	eic_gnt_ic	1=IC が EIC の制御許可
3	valid_req_po	1=PO により IIS に正しい要求がなされた
2	valid_req_dcc	1=DCC により IIS に正しい要求がなされた
1	valid_req_ic	1=IC により IIS に正しい要求がなされた
0	i_source	li_source 信号の状態

a d. iis_err_int

表 1.29 iis_err_int レジスタフィールド

フィールド (ビット)	名称	説明
[31:16]	interrupt	割込み条件
[15:0]	error	エラー条件 ビット 3 = ディスエーブル中に IIS に I バス要求がなされた ビット 2 = 不合法なコプロセッサの位置 から要求があった 画素オーガナイザから ビット 1 = 不合法なコプロセッサの位置 から要求があった データキャッシュコントローラから ビット 0 = 命令コントローラから不合法 なコプロセッサの位置から要求があった

a e. iis_err_int_en

表 1.30 iis_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
[31:16]	int_mask	割込みマスク
[15:0]	err_mask	エラーマスク ビット 3 = ディスエーブル中に IIS に I バス要求がなされた ビット 2 = 不合法なコプロセッサの位置 から要求があった 画素オーガナイザから

		ビット 1 = 不合法なコプロセッサの位置 から要求があった データキャッシュコントローラから ビット 0 = 命令コントローラから不合法 なコプロセッサの位置から要求があった
--	--	--

a f. iis_ic_addr

表 1.31 iis_ic_addr レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	ic_addr	いま IC によって要求されたデータのアドレス

ag. iis_dcc_addr

表 1.32 iis_dcc_addr レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	dcc_addr	いま DCC によって要求されたデータのアドレス

ah. iis_po_addr

表 1.33 iis_po_addr レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	po_addr	いま PO によって要求されたデータのアドレス

ai. iis_burst

表 1.34 iis_burst レジスタフィールド

フィールド (ビット)	名称	説明
[13:0]		Reserved
[21:16]	po_burst	PO からのバースト長
[13:8]	dcc_burst	DCC からのバースト長
[5:0]	ic_burst	IC からのバースト長

aj. iis_base_addr

表 1.35 iis_base_addr レジスタフィールド

フィールド (ビット)	名称	説明
[31:12]	the coprocessor_base [19:0]	メモリマップにおけるコプロセッサ対象メモリのベースアドレス。コプロセッサメモリはページ境界に割当てられる
[11:0]		Reserved

ak. iis_test

表 1.36 iis_test レジスタフィールド

フィールド (ビット)	名称	説明
[31:4]	test_mode[3:0]	Reserved
[3:0]		0=IC->EIC, PO->LMC 0=IC->EIC, DCC->LMC 0=PO->EIC, IC->LMC 0=PO->EIC, DCC->LMC 0=DCC->EIC, IC->LMC

		0=DCC->EIC, PO->LMC
--	--	---------------------

外部インターフェースコントローラレジスタ

a1. eic_cfg

表 1.37 eic_cfg レジスタフィールド

フィールド (bits)	名称	説明
0	reserved	
1	enable	全てのEICの動作を、PCIと汎用バスのマスタとしてイネーブルにする
2	bypass	
3	reserved	EICのバイパスモードの1つ或はそれ以上の動作をイネーブルにする
5:4	pci_arb	PCIバスの裁量 00=公正 01=IBus優先 10=RBus優先 11=公正
6	pci_ibus_min_1 en	
7	pci_rbus_min_1 en	
8	pci_mrl_ram_en	PCIメモリの読出しラインとメモリリード多重モードをイネーブルにする
10:9	gen_arb	
11	gen_clk	0=汎用バスクロックをclkの1/4にする 1=汎用バスクロックをclkの1/2にする

21:12	mmn_mask	MMU で使用されるページサイズを決定するマスクビット
29:24	hash_func	ページテーブルルックアップで使用されるハッシュ機能を決定する
30	hash_bypass	ハッシュ機能が完全にバイパスされていると、ページテーブルインデックスは過疎Uページ番号の下から13ビットとなる (ページサイズに無関係)
31	mmu_bypass	全ての要求に対して MMU をヌルマッピングさせる

am. eic_stat

表 1.38 eic_stat レジスタフィールド

フィールド (bits)	名称	説明
0	ibus_pci_act	Ibus から PCI バスへのアクティブ要求を示す
1	rbus_pci_act	Rbus から PCI バスへのアクティブ要求を示す
2	mmu_act	MMU が動作中を示す
3	gen_act	汎用バスへのアクティブ要求を示す
4	ibus_pci_gnt	Ibus が PCI バスに許可される
5	rbus_pci_gnt	Rbus が PCI バスに許可される
6	mmu_pci_gnt	MMU サービスが PCI バスに許可される
31:4	reserved	

an. eic_err_int

eic_err_int レジスタのエラー及び割り込みビットは EIC のみによって設定でき、ソフトウェアのみによってリセットできる。通常のエラー及び割り込みビットはそのビットに 1 を書き込むことでリセットされ

る。PCI コンフィギュレーションレジスタビットのコピーであるエラービットは PCI コンフィギュレーションレジスタに書き込むことでクリアされなければならない。すなわち、eic_err_int のコピーは何も影響しない。

表 1.39 eic_err_int レジスタフィールド

フィールド (bits)	名称	説明
0	page_fault	ページフォールトエラーによりセットされる
1	prot_fault	保護フォールトエラーによりセットされる
2	gen_err	少なくとも1つの gen_clk サイクルの間、汎用バスエラーが発生するとセットされる
3	gen_addr_err	汎用バスバースト要求が汎用バスのメモリ領域を出るとセットされる
4	iwd_err	ディスエーブルの時の要求。EIC が Rbus 或は Ibus 要求をイネーブルビットが0のときに受取った時にセット。
5	target_abort_err	PCI 構成スペースにおいて受取った TargetAbort ビットをコピーする
6	master_abort_err	PCI 構成スペースにおいて出力された TargetAbort ビットをコピーする
7	reserved	
8	data_parity_exp	PCI 構成スペースにおいて検知したデータパリティエラービットをコピーする
9	gen_exp	少なくとも1つの gen_clk サイクルの間、gen_cxp1 ピンに出力されるとセットされる

23:10	reserved	
24	gen_int	少なくとも1つの gen_clk サイクルの間、gen_int1 ピンに出力されるとセットされる
31:25	reserved	

a0. eic_err_int_en

表 1.40 eic_err_int レジスタフィールド

フィールド (ビット)	名称	説明
0	page_fault	対応するエラービットをイネーブルにして c_err を出力する
1	prot_fault	対応するエラービットをイネーブルにして c_err を出力する
2	gen_err	対応するエラービットをイネーブルにして c_err を出力する
3	gen_addr_err	対応するエラービットをイネーブルにして c_err を出力する
4	rwd_err	対応するエラービットをイネーブルにして c_err を出力する
5	target_abort _err	対応するエラービットをイネーブルにして c_err を出力する
6	master_abort _err	対応するエラービットをイネーブルにして c_err を出力する
7	reserved	
8	data_priorit y_exp	対応するエラービットをイネーブルにして c_exp を出力する
9	gen_exp	対応するエラービットをイネーブルにして c_exp を出力する

23:10	reserved	
24	gen_int	対応するエラービットをイネーブルにして c_int を出力する
31:25	reserved	

a p. eic_test

表 1.41 eic_test レジスタフィールド

フィールド (ビット)	名称	説明
31:0	未定義	

a q. eic_pob

表 1.42 eic_pob レジスタフィールド

フィールド (ビット)	名称	説明
0	gen_pob0	汎用バスのプログラマブル出力ビ ン pub0 の値
1	gen_pob1	汎用バスのプログラマブル出力ビ ン pub1 の値

a r. eic_high_addr

表 1.43 eic_dual_addr レジスタフィールド

フィールド (ビット)	名称	説明
31:0	high_addr	コプロセッサデータ構造が存在している ホストメモリ (2重アドレスサイクル) の 4Gバイト領域のオフセット

a s. eic_wtlb_v

表 1.44 eic_wtlb_v レジスタフィールド

フィールド (ビット)	名称	説明
1:0	pt_flags	書込まれる TLB エントリーに関連し たフラグ
11:2	reserved	
31:12	vpn	書込まれる或は無効化される仮想ペ ージ番号

a t. eic_wtlb_p

表 1.45 eic wtlb p レジスタフィールド

フィールド (ビット)	名称	説明
0	inv_all	TLB 全体を無効にする。TLB の動作後、EIC はこのビットをクリアする
1	inv_entry	仮想ページ番号が特定されると TLB エントリは無効にされる。TLB の動作後、EIC はこのビットをクリアする
2	write_lru	特定された仮想ページ番号、物理ページ、制御ビットで LRU 位置に TLB エントリを書込む。TLB の動作後、EIC はこのビットをクリアする
3	write	特定された仮想ページ番号、物理ページ、フラグビットを有するエントリ番号で特定された TLB エントリを書込む。TLB の動作後、EIC はこのビットをクリアする
7:4	ent_num	エントリ書き込み動作で書込まれたエントリ番号
31:12	ppn	書込まれるべき物理ページ番号

a u. eic_mmuv レジスタフィールド
注: このレジスタの値は、MMU がページフォールトエラーあるいは MMU から PCI バスのエラーにより無効でないなら、いつでも変更可能である。

表 1.46 eic mmu v レジスタフィールド

フィールド (ビット)	名称	説明
3:0	mmu_lru	現在の LRU 位置
4	mmu_hit	最後の TLB アクセスページテーブルアクセス無しにヒットすると、mmu_hit がセットされる
5	mmu_retry_hit	最後の TLB アクセスページテーブルアクセスの後でヒットすると、v_mmu_retry_hit がセットされる
11:6	Reserved	
31:12	mmu_lvpn	変換のために MMU に送られる最も最近の仮想ページ番号

a v. eic_mmu_p レジスタフィールド
注: このレジスタの値は、MMU がページフォールトエラーあるいは MMU から PCI バスのエラーにより無効でないなら、いつでも変更可能である。

表 1.47 eic mmu p レジスタフィールド

フィールド (ビット)	名称	説明
31:0	mmu_lpta	MMU によりフェッチされたもっとも最近のページテーブルの物理アドレス

a w. eic_ip_addr レジスタフィールド
注: このレジスタの値は IBD が I Bus から PCI バスに転送されたときに有効になる。

スへのエラーによって無効でないならいつでも変更可能である。

表 1.48 eic_ip_addr レジスタフィールド

フィールド (ビット)	名称	説明
31:0	libpa	PCIBus にアクセスするもっとも最近の IBus の物理アドレス

a x. eic_rp_addr

スへのエラーによって無効でないなら、いつでも変更可能である。

注: このレジスタの値はRBRがRBusからPCIバ

表 1.49 eic_rp_addr レジスタフィールド

フィールド (ビット)	名称	説明
31:0	lrbpa	PCIBus にアクセスするもっとも最近の RBus の物理アドレス

a y. eic_ig_addr 注: このレジスタの値はGBCが汎用バスのエラーによって無効でないなら、いつでも変更可能である。

表 1.50 eic_ig_addr レジスタフィールド

フィールド (ビット)	名称	説明
31:0	lrbga	汎用busにアクセスするもっとも最近の IBus のアドレス

a z. eic_rg_addr 注: このレジスタの値はGBCが汎用バスのエラーによ

表 1.51 eic_rg_addr レジスタフィールド

フィールド (ビット)	名称	説明
31:0	ixbga	汎用バスにアクセスするもっとも最近の RBus のアドレス

PCIバスコンフィギュレーション空間のエイリアス
16ワードからなるPCIバスコンフィギュレーション
空間は0xc0から0xcfまでのアドレスで示される

レジスタにエイリアスされている。
ローカルメモリコントローラレジスタ
b a. lmi_cfg

表 1.52 lmi_cfg レジスタフィールド

フィールド (ビット)	名称	説明
[31:24]	ref_interval	4 clk 周期の乗算におけるレフレッシュ間隔
[23:22]	ro_prio	RO アクセス優先度 (3=最高)
[21:20]	pic_prio	PIC アクセス優先度 (3=最高)
[19:18]	ic_prio	IC アクセス優先度 (3=最高)
[17:16]	iis_prio	IIS アクセス優先度 (3=最高)
[15:13]	rearb_interval	ワードにおける再調停期間 (2^n ワード)
[12]	mem_enable	メモリーネーブル (1=メモリー装着済み)
[11]	banks	接続されている DRA バンクの数 (0=1, 1=2)
[10]	iis_flush	IIS 先取りデータのフラッシュ (自己クリア)
[9]	iis_prefetch	IIS の先取りイネーブル
[8]	refresh_en	DRAM の自動リフレッシュイネーブル (CBR)
[7]	scanless	継目のない SDRAM のページフォールトイネーブル
[6]		Rservcd
[5:4]	bypass_mode	バイパスモード選択: 00=Rbus 01=Abus 書込み 10=Abus リード 11=Ibus
[3]	s	自己テストモード
[2]	B	バイパスイネーブル
[1]	E	イネーブルモジュール
[0]		Reserved

このレジスタは LMC の処理モードとパラメータを決定するのに用いられる多くのコンフィギュレーションビットと制御ビットを含む。sdr_am_1 ピンがハイの時

SDRAM 処理を特別に参照するビットは全く影響を持たない。このレジスタは clk_in の周波数が 80 MHz のとき 3.2 マイクロ秒のリフレッシュ間隔である

ようなリセット値0x20000100をもつ。すべての特別なモードや機能は電源投入時には無効であり、すべてのアクセス権限は等しく0に設定される。リフレッシュはリセット時に有効であるが、ほかのモジュールは

無効 (E=0) である。リフレッシュはEビットに影響されない。

b b. lmi_stat

表 1.53 lmi_stat レジスタフィールド

フィールド (ビット)	名称	説明
[31]	ro_ca	RO モジュールサイクルアクティブ
[30]	pic_ca	PIC モジュールサイクルアクティブ
[29]		rcserved
[28]	iis_ca	IIS モジュールサイクルアクティブ
[27]	ro_cp	RO モジュールサイクルペンディング
[26]	pic_cp	PIC モジュールサイクルペンディング
[25]		reserved

[24]	iis_cp	IIS モジュールサイクルペンディング
[23:22]		Reserved
[21:16]	stateA	内部コントロールステートA
[15:13]		Reserved
[13:8]	stateB	内部コントロールステートB
[7]	rd_active	DRAMの読出し
[6]	wr_active	DRAMの書込み
[5]	rf_active	リフレッシュ進行中
[4]	rf_pending	リフレッシュ・ペンディング
[3]	iis_pre_active	IIS 先取りアクティブ
[2:1]		Reserved
[0]	sdrarm	sdrarm 1 入力ピンの状態

ステータスレジスタはマシン内部の情報と同様にモジュールのアクティブや未決定ビットからなる。ステートマシンはCBusインターフェースの2倍のクロックで駆動されており、従って最新の80MHzクロック2サイ

クルそれぞれの状態情報を保持するには2フィールド必要である。

b c. lmi_err_int

表 1.54 lmi_err_int レジスタフィールド

フィールド (ビット)	名称	説明
[31:24]	interrupt	割り込みステータスビット
[23:8]	exception	例外ステータスビット
[7:0]	error	エラーステータスビット

エラーと割り込みのステータスレジスタは割り込み、例外、エラー状態の情報を保持する。レジスタは読み書きでき、読み込みはステータス情報を返し、特定ビットへ

の1の書き込みはそのビットをリセットする。0の書き込みはそのビットに対して全く影響を持たない。

表 1.55 割り込みビットの定義

ビット	説明
24	レフレッシュ割り込み。各リフレッシュ間隔 (64 ms) で発生する
25	後続のリフレッシュ要求に先行してリフレッシュしない

表 1.56 エラービットの定義

ビット	説明
0	DRAMがイネーブルでない時に DRAM にアクセスした
1	モジュールがディスエーブルのときに DRAM にアクセスした
2	Rbus アドレスエラー
3	Abus アドレスエラー
4	Cbus アドレスエラー
5	Ibus アドレスエラー

このレジスタはリセット値 0x00000000 を持つ。状態を変更できない。なくてはならず、これは割り込み及びエラーが発生していないことを示す。予約ビットは常に0であり決して状

b d. lmi_err_int_en レジスタ

表 1.57 lmi_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
[31:24]	int_mask	割り込みマスクビット
[23:8]	exp_mask	例外マスクビット
[7:0]	err_mask	エラーマスクビット

エラー、例外、割り込み有効レジスタはエラー、例外割り込み信号の有効、無効の選択に用いられる。レジスタは読み書きできる。このレジスタは lmi_err_int レジスタ内のエラー、例外、割り込みそれぞれに基づいて、ビット単位で有効化するのに用いられる。このレジスタのビットと lmi_err_int レジスタのビットとの間には1対1の対応がある。もし lmi_err_int_en レジスタの特定のビットがハイにな

ったら lmi_err_int レジスタの対応するビットが有効になり、それがハイであるならば、LMC モジュールエラー、例外あるいは割り込み信号、c_err、c_exp、あるいは c_int が発生できる。もし lmi_err_int_en レジスタの特定のビットがクリアされたら lmi_err_int レジスタの対応するビットが無効になり、c_err、c_exp あるいは c_int を発生させることはできない。

LMCには例外はないので、このレジスタの `exp_mask` ビットは全く影響せず、すべて予約である。このレジスタのリセット値はすべてのエラー及び割り込み源を無効にする `0x00000000` である。使用されな

いビットは常に0であり、ハイにセットすることはできない。

b e. `lmi_dcfg`

表 1.58 `lmi_dcfg` レジスタフィールド

フィールド (ビット)	名称	説明
[31:29]	<code>row_bits</code>	DRAM の行 (row) アドレスビット数 (n+6)
[28:26]	<code>column_bits</code>	DRAM の列 (column) アドレスビット数 (n+6)
[25]	<code>edo_dram</code>	EDO 対応の DRAM (0=FP, 1=EDO)
[24:22]	<code>precharge_enable</code>	プリチャージ可能なアドレスビット (n+5)
[21:19]	<code>precharge_bank</code>	プリチャージ・バンクアドレスビット (n+5)
[18:17]	<code>cas_lo</code>	CAS の低クロックサイクル数 (CAS パルス幅) (1 乃至 4, o=1)
[16]	<code>cas_hi</code>	CAS の高クロックサイクル数 (CAS のプレチャージ) (1 乃至 2, o=1)

[15:13]	<code>ras_lo</code>	RAS の最小低クロックサイクル (1 乃至 8, o=1)
[12:11]	<code>ras_hi</code>	RAS の最小高クロックサイクル (RAS のプレチャージ) (1 乃至 4, o=1)
[10:9]	<code>ras_cas</code>	クロックサイクルにおける RAS から CAS への遅れ (2~6, o=2)
[8:7]	<code>cas_lat</code>	クロックサイクルにおける SDRAM CAS latency (潜在): 0, 1=1クロック 2=2クロック 3=3クロック
[6:0]		reserved

このコンフィギュレーションレジスタは DRAM チップを使用する場合のサイズやコンフィギュレーションを決定する設計パラメータを保持する。このレジスタはす

べてのタイミング制限の値を最大値にするようリセット値 `0x0007ff80` を保持する。

b f. `lmi_mode` レジスタ

表 1.59 lmi_mode レジスタフィールド

フィールド (ビット)	名称	説明
[31:14]		Reserved
[13]	initilize	SDRAM の初期化 (プログラムモードレジスタ, 自己クリア)
[12:0]	sdram_mode	SDRAM モードビット (SDRAM コマンドを初期化することによりアドレスピンから SDRAM モードレジスタに書き込む)

このコンフィギュレーションレジスタは初期化処理のシーケンスでハードウェアを用いたフルページに設定する。あらゆるリセットの後、もし `sdram_1` ピンがローであれば、SDRAM モードレジスタを初期的にプログラムするために、初期化ビットはセットされる。モードレジスタの書き込み実行後、このビットは自動的にゼロにクリアされる。

周辺インターフェースレジスタ
b.g. `pic_cfg` レジスタ

このコンフィギュレーションレジスタは初期化処理のシーケンスでハードウェアを用いたフルページに設定する。あらゆるリセットの後、もし `sdram_1` ピンがローであれば、SDRAM モードレジスタを初期的にプログラムするために、初期化ビットはセットされる。モードレジスタの書き込み実行後、このビットは自動的にゼロにクリアされる。

周辺インターフェースレジスタ
b.g. `pic_cfg` レジスタ

表 1.60 pic_cfg レジスタフィールド

フィールド (ビット)	名称	説明
0	reserved	
1	enable	全てのPIC動作をイネーブルにする
2	bypass	
3	reserved	
4	big_endian	全てのバイトをパッキング/アンパッキングして <code>big_endian</code> の順を使用する。即ち、最初はビット 31:24で、後はビット 7:0
5	video mode	0=ビデオ出力 1=ビデオ入力
7:6	reserved	
8	gate_out_c lock	ビデオ出力モードで使用。出力クロックを <code>vclk_en_in_1</code> に関して反転させる
9	inv_out_cl ock	ビデオ出力モードで使用。出力クロックを <code>vclk_in</code> に関して反転させる

10	use_default_data	ビデオ出力モードで使用。このビットがセットされていると、default_dataレジスタビットの値が、データが有効でないサイクルの間、出力データピンに出力される。一方、もし利用できれば、次の有効データバイトが出力される
11	default_enable_source	ビデオ出力モードで使用。もしuse_default_dataがイネーブルであれば、default_dataデータを使用する
12	data_enable_source	ビデオ出力モードで使用。1 のとき、vdata_en_in_1がサンプルされ、出力されているときvdata_en_out_1が出力される。一方、vclk_en_in_1がサンプルされ、出力されているときvdata_en_out_1が出力される。
13	del_data	ビデオ出力モードで使用。vdata_en_out_1に関して出力データを1クロックサイクル遅延させる
14	gate_with_rdy	ビデオ出力モードで使用。出力クロックをvrdy_1でゲートする
15	reserved	
16	forward_active_disable	出力FIFOのデータによりc_activeの出力をディスエーブルにする
17	reverse_active_disable	入力FIFOのデータによりc_activeの出力をディスエーブルにする
18	forward_priority	双方向モードで順方向の転送を優先させる

19	reverse_priority	双方向モードで逆方向の転送を優先させる
20	forward_invert_data	反転出力データ信号
21	reverse_invert_data	反転出力データ信号
22	forward_invert_control	反転出力制御信号
23	reverse_invert_control	反転入力制御信号
30:24	cbus_timer	入力或は出力データストリーム CBus サイクルがエラー信号の前にストールされる最大時間。8クロックサイクル乗数で決められる
31	cbus_timer_disable	CBus 体目の動作ディスエーブル

bh. pic_stat

表 1.61 pic_stat レジスタフィールド

フィールド (ビット)	名称	説明
0	abus_act	ABus のトランザクションがペンディング 或は実行中にセットされる
1	rbus_act	RBus のトランザクションがペンディング 或は実行中にセットされる
2	reverse_act	逆方向のデータ転送がアクティブの時セット

3	cent_cfg	外部 cent_cfg ピンのコピー
4	forward_full	PIC の出力 FIFO がフルの時にセット
5	forward_subword	PIC の出力 FIFO に 0 以上 4 バイト以下のスペースがある時にセット
6	forward_empty	PIC の出力 FIFO が空の時にセット
7	reserved	
8	reverse_full	PIC の入力 FIFO がフルの時にセット
9	reverse_empty	PIC の入力 FIFO が空の時にセット
10	reverse_subword	PIC の入力 FIFO に 0 以上 4 バイト以下のスペースがある時にセット
15:11	reserved	
16	pic_plh	pic_plh ピンの状態
17	pic_ack_1	pic_ack_1 ピンの状態
18	pic_busy	pic_busy ピンの状態
19	pic_perror	pic_perror ピンの状態
20	pic_select	pic_select ピンの状態
21	pic_fault_1	pic_fault_1 ピンの状態
31:24	pic_data_in	pic_data_in ピンの状態

bi. pic_err_int のみによってリセットされる。それぞれのビットは 1 を
 pic_err_int レジスタのエラーおよび割り込み書き込むことでリセットされる
 みビットは PIC のみによりセットされ、ソフトウェア

表 1.62 pic_err_int レジスタフィールド

フィールド (ビット)	名称	説明
----------------	----	----

0	forwad_err	順方向転送エラー。PIC が入力専用モードの時に出力データを受取るとセット
1	rwd_err	デイスエーブルエラーの時の要求。PIC がデイスエーブルでRBus トランザクションの要求、或はいずれかのレジスタの初期化動作 (ABus 転送、セントロニクス命令等) を受取ったときにセット
2	timeout_err	周辺タイムアウトエラー
8	reverse_exp	逆転送エラー。PIC が出力専用モードの時にソフトウェアがコプロセッサに逆方向転送しようとするときセット
9	cbus_exp	入出力データストリームの CBus サイクルがタイムアウトになるとセットされる。このビットがセットされると、入出力データストリームの CBus サイクルが逆として処理される
24	rev_comp_int	逆方向転送においてバイトカウント限界でセット
25	rev_data_int	逆方向転送空のデータがあつてその宛先を示すバストランザクションがアクティブでないときにセット
26	abus_comp_int	abus 転送が完了時にセット
27	timer_int	タイマ0或は1の動作が終了した時にセット
28	comp_fault_int	セントロニクス準拠モードで pic_fault_1 信号が出力されたときにセット

29	comp_perror_ int	セントロニクス準拠モード でpic_perror 信号が出力されたときに セット
30	ecp_rev_req_ int	セントロニクス ECP モードで周辺が逆デ ータ転送であるときにセット
31	no_periphera l_int	周辺割込みなし

b j. pic_err_int_en

表 1.63 pic_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
0	forward_err	対応する割込みビットをイネーブルにし てc_intを出力させる
1	rwd_err	対応する割込みビットをイネーブルにし てc_intを出力させる
2	timeout_err	対応する割込みビットをイネーブルにし てc_intを出力させる
8	reverse_exp	対応する割込みビットをイネーブルにし てc_intを出力させる
9	cbus_exp	対応する割込みビットをイネーブルにし てc_intを出力させる
24	rev_comp_int	対応する割込みビットをイネーブルにし てc_intを出力させる
25	rev_data_int	対応する割込みビットをイネーブルにし てc_intを出力させる
26	abus_comp_int	対応する割込みビットをイネーブルにし てc_intを出力させる

27	timer_int	対応する割込みビットをイネーブルにして c_int を出力させる
28	comp_fault_int	対応する割込みビットをイネーブルにして c_int を出力させる
29	comp_perror_int	対応する割込みビットをイネーブルにして c_int を出力させる
30	ecp_rev_req_int	対応する割込みビットをイネーブルにして c_int を出力させる
31	no_peripheral_int	対応する割込みビットをイネーブルにして c_int を出力させる

b k. pic_abus_cfg

表 1.64 pic_abus_cfg レジスタフィールド

フィールド (ビット)	名称	説明
23:0	ab_count	IMC へ/から転送の残りバイト数。最初はソフトウェアによりセットされ、各バイトが転送される度に PIC により 1 される
27:24	ab_byte_en	ABus 転送で使用されるバイトレーン
28	ab_type	1=転送が読み込み (周辺ポートから) 0=転送が書き込み (周辺ポートへ)
29	ab_the_coproccs-sort	コプロセッサによる ABus 転送。ABus 転送が終了するとクリアされる。ソフトウェアによりクリアされると、ABus 転送が終了する
31:30	reserved	

b l. pic_abus_addr

表 1.65 pic_abus_addr レジスタフィールド

フィールド (ビット)	名称	説明
31:0	ab_addr	次の ABus 転送がコプロセッサによる場合のバイトアドレス

b m. pic_cent_cfg

pic_cent_cfg レジスタはセントロニクスモードが有効の場合に、すべてのインターフェースの局面

を制御する読み込み/書き込み信号及び読み込み専用ステータス信号を含んでいる。

表 1.66 pic_cent_cfg レジスタフィールド

フィールド (ビット)	名称	説明
2:0	cen_cmd	セントロニクスコマンド: 000=コンパチブルに復帰 001=ニブル要求 010=ニブルを使用してデバイスID 転送 011-100=意味なし 101=ECP 要求 110=ECP を使用してデバイスID転送 111=失敗
3	cen_exe	プログラムされたセントロコマンドのコプロ セッサによる実行このビットはこのセントロ コマンドが終了するとリセットされる
4	cen_sts	現セントロモード及び最後の協議に基づく動 作を示す読出し専用ビット。1はコマンドが成 功したとき、0は失敗したときを示す

7:5	cen_mode	現セントロモード及び最後の協議に基づく動作を示す読出し専用ビット 000=コンパチブル 001=ニブルがイネーブルでコンパチブル 010=ニブルを使用してデバイス ID を送る 011=ニブルモード 100=reserved 101=ECP 順方向モード 110=ECp を使用してデバイス ID を送る 111=ECP 逆方向モード
8	cen_direct	セントロニクス制御をバイパス，このレジスタ書込み或は読み出すことにより，ソフトウェアが直接出力ビットを制御する
9	cen_host_rec_en	ECP モードでホスト再生機構をイネーブル
15:10	reserved	
23:16	cen_tim	クロックサイクルにおけるタイマ値で，全てのセントロニクスにセットアップされ、保持されるパルス時間
31:24	cen_per_tim	セントロニクスの周辺からの応答を待っている時の64のロットのタイマ値、

bn. pic_cent_dir

表 1.67 pic_cent_dir レジスタフィールド

フィールド (ビット)	名称	説明
0	pic_data_oe	pic_data_oe 信号の直接制御ビット

1	pic_strobe_1	pic_strobe_1 信号の直接制御ビット
2	pic_autofd_1	pic_autofd_1 信号の直接制御ビット
3	pic_init_1	pic_init_1 信号の直接制御ビット
4	pic_selection_1	pic_selection_1 信号の直接制御ビット
5	pic_buf_en_1	pic_buf_en_1 信号の直接制御ビット
6	pic_buf_dir	pic_buf_dir 信号の直接制御ビット
7	reserved	
15:8	pic_data_out	pic_data_out 信号の直接制御ビット
31:16	reserved	

b o. pic_reverse_cfg

表 1.68 pic_reverse_cfg レジスタフィールド

フィールド (ビット)	名称	説明
23:0	rev_count	外部周辺から転送の残余バイト数最初はソフトウェアによりセットされ、各バイトが転送される度に PCI により -1 される
24	rev_use_count	特定のバイト数が転送された後、逆方向転送を終了させる
25	rev_the_coprocessor	コプロセッサによるプログラムされた逆方向転送。rev_use_count ビットがセットされていると、逆方向転送が終了するとクリアされる。ソフトウェアは、いつでも逆方向転送を終了させるためにこのビットをクリアできる
31:26	reserved	

b p. pic_timer0

表 1.69 pic_timer0 レジスタフィールド

フィールド (ビット)	名称	説明
31:0	timer0	最初のデータタイムアウトでのタイマ値 (1 クロックの単位)

b q. pic_timer1

表 1.70 pic_timer1 レジスタフィールド

フィールド (ビット)	名称	説明
31:0	timer1	最初のデータからデータへのタイムアウトでのタイマ値 (1 クロックの単位)

データキャッシュコントローラレジスタ

bs. dcc_cfg2

br. dcc_cfg1

表 1.71 dcc_cfg1 レジスタフィールド

フィールド (ビット)	名称	説明
1	E	0=ディスエーブル 1=イネーブル
2	B	0=バイパスしない 1=バイパスする

表 1.72 dcc_cfg2 レジスタフィールド

フィールド (ビット)	名称	説明
----------------	----	----

0	Autoinv	0=自動無効化モードでない (リセット時の値) 1=各命令の最後で自動無効化キャッシュ
1	Autofill	0=自動無効化モードでない (リセット時の値) 1=dcc_addr で特定されたアドレスに対応する自動フルキャッシュ。このビットは自己クリア
2	Lock	0=データフェッチの後、キャッシュRAMにデータが書き込まれる (リセット時の値) 1=データフェッチの後、キャッシュRAMにデータが書き込まれない
3	always_hit	0=タグメモリと有効ビットがチェックされる (リセット時の値) 1=データがいつも有効であってキャッシュが残っていると仮定する
4	always_miss	0=タグメモリと有効ビットがチェックされる (リセット時の値) 1=データがキャッシュに残っていない仮定
[10:8]	mode	現モードがキャッシュ動作: 0=ランダムアクセスノーマルモード (リセット時の値) 1=JPEG 符号化 2=低速 JPEG 復号 3=信号チャネルカラー空間変換 4=多重チャネルカラー空間変換 5=行列演算 6=64ビットモード 7=キャッシュの全エントリを無効にする。DCCは全てのライン有効ビットがクリアされるとモード1に変える。このフィールドは各命令の終了時リセットされる

11	cache_miss_inst	現在のキャッシュ動作 0=タグメモリと有効ビットがチェックされる (リセット時の値) 1=データがキャッシュに残っていない仮定 このビットは各命令の終了時リセットされる
----	-----------------	---

bt. dcc_stat

表 1.73 dcc_stat レジスタフィールド

フィールド (ビット)	名称	説明
[3:0]	error[3:0]	エラー条件: ビット0=モジュールディスエーブル時のキャッシュ またはタグメモリへのR/W動作
[p:q]	state[7:0]	内部状態条件。

bu. dcc_err_int

表 1.74 dcc_err_int レジスタフィールド

フィールド (ビット)	名称	説明
0	error	モジュールディスエーブル時のキャッシュ またはタグメモリへのR/W動作

bv. dcc_err_int_en

表 1.75 dcc_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
0	error_en	対応エラーイネーブル

bw. dcc_lv0

表 1.76 dcc_lv0 レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	lv0[31:0]	ライン31から0までの有効ビット

bx. dcc_lv1

表 1.77 dcc_lv1 レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	lv1[31:0]	ライン63から32までの有効ビット

by. dcc_lv2

表 1.78 dcc_lv2 レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	lv2[31:0]	ライン95から64までの有効ビット

bz. dcc_lv3

表 1.79 dcc_lv3 レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	lv3[31:0]	ライン127から96までの有効ビット

c a. dcc_addr

表 1.80 dcc_addr レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	dcc_addr	特別なアドレッシングモードで使用されるベースアドレスレジスタ。このレジスタの別のビットは動作モードに関するベンディングに使用される

c b. dcc_raddrb

表 1.81 dcc_raddrb レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	dcc_raddrb	オペランドオーガナイザBのアドレスの状態

c c. dcc_raddrc

表 1.82 dcc_raddrc レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	dcc_raddrc	オペランドオーガナイザCのアドレスの状態

c d. dcc_test

表 1.83 dcc_test レジスタフィールド

フィールド (ビット)	名称	説明
0	drive_odata	test_mode ビットが0にセットされているときのみ有効 0=オペランドオーガナイザデータベースを駆動しない (リセット時の値) 1=オペランドオーガナイザデータベースを駆動する
1	o_ackb	test_mode ビットが0にセットされているときのみ有効。DCC がテストモードでdrive_odata がセットされているときのみ使用可能。このビットは自己クリア 0=o_ackb を出力しない (リセット時の値) 1=1クロックの間、o_ackb を出力する

2	o_ackc	test_mode ビットが0にセットされているときのみ有効。DCC がテストモードで drive_odata がセットされているときのみ使用可能。このビットは自己クリア 0=o_ackc を出力しない (リセット時の値) 1=1 クロックの間、o_ackc を出力する
[5:3]	test_mode	0=odata バスモードを駆動する (リセット時の値) 1=00B アドレスを IIS に出力、i_oe=0 2=00C アドレスを IIS に出力、i_oe=0 3=IIS データを o_datab[31:0] に出力、i_oe=1 3=IIS データを o_datac[31:0] に出力、i_oe=1

オペランドオーガナイザレジスタ オペランドオーガナイザ用のレジスタはここに記述されている。

イザレジスタには同様の2つのオペランドオーガナイザが存在する: オペランドオーガナイザBとオペランドオーガナイザCである。これらの2つのオペランドオーガナイザ用のレジスタはここに記述されている。
ce. oon_cfg (oob_cfg=0x70, ooc_cfg=0x80)

表 1.84 oon_cfg レジスタフィールド

フィールド (ビット)	名称	説明
[31:9]	operate	reserved
[8]		OO (格命令の終わりに自己クリア) のための コプロセッサビット
[7:2]		reserved
[1]	E	0=ディスエーブル 1=イネーブル
[0]		reserved

cf. oon_stat (oob_cfg=0x71, ooc_cfg=0x81)

表 1.85 oon_stat レジスタフィールド

フィールド (ビット)	名称	説明
[31:24]	state[7:0]	reserved
[23:16]		内部状態条件: ビット0=OOアクティブ ビット1=OOストール (シーケンスモードのみ) ビット2=FIFO 空 ビット3=FIFO フル
[15:0]		reserved

cg. oon_err_int (oob_err_int=0x72, err_int=0x82)

表 1.86 oon_err_int レジスタフィールド

フィールド (ビット)	名称	説明
[15:0]	error	エラー条件: ビット0=ディスエーブルの時にOOによりO Bus 要求が受信された ビット1=MDP と JPEG 符号化要求データが同時発生

ch. oon_err_int_en (oob_err_int_en=0x83)
r_int_en=0x73, err_int_en=

表 1.87 oon_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
[15:0]	error	エラー条件: ビット0=ディスエーブルの時にOOによりO Bus 要求が受信された ビット1=MDP と JPEG 符号化要求データが同時発生

ci. oon_dmr (oob_dmr=0x74,
ooc_dmr=0x84)

表 1.88 oon_dmr レジスタフィールド

フィールド (ビット)	名称	説明
[31 : 30]	ls3	バイト3のレインスワップ: 0=バイト0から 1=バイト1から 2=バイト2から 3=スワップなし
[29 : 28]	ls2	バイト2のレインスワップ: 0=バイト0から 1=バイト1から 2=スワップなし 3=バイト3から
[27 : 26]	ls1	バイト1のレインスワップ: 0=バイト0から 1=スワップなし 2=バイト2から 3=バイト3から
[25 : 24]	ls0	バイト0のレインスワップ: 0=スワップなし 1=バイト1から 2=バイト2から 3=バイト3から
[23 : 20]	subst[3:0]	代替イネーブル: 0=このバイトの入替え動作を行わない 1=oon_subst に記憶された値をこのバイトに使用
[19 : 15]	replicate	応答カウント: 生成される付加データの数を示す

[14:12]	normalise	正規化要素: 0=チャンネル当たり1ビット 1=チャンネル当たり2ビット 2=チャンネル当たり4ビット 3=チャンネル当たり8ビット 4~7=チャンネル当たり16ビット
[11]		reserved
[10:8]	bo[2:0]	対象ビットのバイト内のビットオフセット
[7]	p	外部フォーマット: 0=入力オブジェクトがパックされていないバイト 1=入力オブジェクトがパックされたストリーム
[6:5]	if	内部フォーマット: 0=画素 1=バックされていないバイト 2=バックされたバイト 3=その他
[4:3]	cc	入力チャンネルカウント: 0=4 アクティブチャンネル 1=1 アクティブチャンネル 2=2 アクティブチャンネル 3=3 アクティブチャンネル
[2]	L	0=oo_said レジスタへのデータ 1=ダイレクトアドレッシング

[1:0]	what	アドレッシングモード: 0=バイパス 1=順次 2=タイリング 3=一定データ
-------	------	---

cj. oon_subst (oob_subst=0 ooc_cdp=0x86)
x75, ooc_subst=0x85)

表 1.89 oon_subst レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	subst	代替データ値

ck. oon_cdp (oob_cdp=0x76,

表 1.90 oon_cdp レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	cdp	現データポインタ

cl. oon_len (oob_len=0x77,
ooc_len=0x8

表 1.91 oon_len レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	len	入力オペランドストリーム長

cn. oon_said (oob_said=0x7 8, ooc_said=0x88)

表 1.92 oon_said レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	said	コプロセッサ処理のアドレス或はデータ

cn. oon_tile (oob_tile=0x7
9, ooc_tile=0x89)

表 1.93 oon_tile レジスタフィールド

フィールド (ビット)	名称	説明
[31:16]	offset[15:0]	オペランドのオフセット
[15:0]	tile_len[15:0]	タイリング動作のオペランドバイトにおける長さ(length)-1

ピクセルオーガナイザレジスタ

co. po_cfg

表 1.94 po_cfg レジスタフィールド

フィールド (ビット)	名称	説明
[26:24]	burst[2:0]	入力データの最大バースト長の2を基底とした log データバーストはこの値のアドレスに割当てられる
[22:21]	chan	JPEG 圧縮のための入力チャネル数 0=1つのチャネル 2=3 チャネル 3=4 チャネル
[20]	ss	0=サブサンプリングなし 1=サブサンプリング実行
[19]	F	サブサンプリングのためのフィルタオプション: 0=フィルタリングなし (2バイト毎) 1=フィルタリング (平均)
[18:16]		reserved
[15]	dst	PBus データの宛先: 0=MDP 1=JPEG 符号

[14:12]	mode[2:0]	PO 用の動作モード: 0=アイドル 1=順次モード 2=JPEG 圧縮 3=CSC 4~7=畳み込み/画像変換
[11:4]		reserved
[3]	s	0=自己テストモードでない 1=自己テストモード
[2]	B	0=バイパスしない 1=バイパスする
[1]	E	0=ディスエーブル 1=イネーブル
[0]		reserved

c.p. po_stat

表 1.95 po_stat レジスタフィールド

フィールド (ビット)	名称	説明
[31:27]	mvv_miss	0=MUV がミスなし 1=MUV ミス
[25]	po_stall	0=PO がストールしない 1=PO がストール
[24]	po_active	0=PO がアクティブでない 1=PO がアクティブ
[20]	sm_the coprocessor	PO がコプロセッサ状態
[19]	ack_rcvd	ibus がアック受信
[18]	ibus_req_made	ibus 要求モード

[17]	fifo_full	FIFO フル
[16]	fifo_empty	FIFO 空
[15:0]		reserved

c.q. po_err_int

表 1.96 po_err_int レジスタフィールド

フィールド (ビット)	名称	説明
[7:0]	error	エラー条件: ビット 0=ディスエーブルのときに IBus が i_ack を PO に出力 ビット 1=MUVRAM を PO と RO が同時使用 ビット 2=JPEG モードでのデータオーバーフロー

c.r. po_err_int_en

表 1.97 po_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
[7:0]	err_mask	エラーマスク: ビット0=ディスエーブルのときに IBus が i_ack を PO に出力 ビット1=MUVRAM を PO と RO が同時使用 ビット2=JPEG モードでのデータオーバーフロー

cs. po_dmr

表 1.98 po_dmr レジスタフィールド

フィールド (ビット)	名称	説明
[31:30]	ls3	バイト3のレインスワップ: 0=バイト0から 1=バイト1から 2=バイト2から 3=スワップなし
[29:28]	ls2	バイト2のレインスワップ: 0=バイト0から 1=バイト1から 2=スワップなし 3=バイト3から
[27:26]	ls1	バイト1のレインスワップ: 0=バイト0から 1=スワップなし 2=バイト2から 3=バイト3から
[25:24]	ls0	バイト0のレインスワップ: 0=スワップなし 1=バイト1から 2=バイト2から 3=バイト3から
[23:20]	suben[3:0]	バイト代替イネーブル: 0=このバイトに po_subst レジスタからのデータによる入替えを行わない 1=po_subst レジスタ値をこのバイトに使用
[19:15]	replicate[4:0]	応答カウント: 生成される付加データの数を示す

[14:12]	norm_facto r[2:0]	入力データの正規化要因: 0=コンポーネント当たり1ビット 1=コンポーネント当たり2ビット 2=コンポーネント当たり4ビット 3=コンポーネント当たり8ビット 4~7=コンポーネント当たり16ビット
[11]		reserved
[10:8]	bo[2:0]	バイト内のビットオフセット ビットアドレッシングはbig endian
[7]	P	外部フォーマット: 0=入力データがパックされていないバイト 1=入力データがパックされたストリーム
[6:5]	if[1:0]	内部フォーマット: 0=画素 1=パックされていないバイト 2=パックされたバイト 3=その他
[4:3]	cc[1:0]	入力ストリームの入力チャネルカウント: 0=4 アクティブチャネル 1=1 アクティブチャネル 2=2 アクティブチャネル 3=3 アクティブチャネル
[2]	L	0=オペランド (short フォーマット) 1=ダイレクトアドレッシング (long フォーマット)

[1:0]	what[1:0]	DMJ のアドレッシングモード: 0=バイパス 1=順次 2=タイリング 3=一定データ
-------	-----------	--

ct. po_subst

表 1.99 po_subst レジスタフィールド

フィールド (ビット)	名称	説明
31:0	subst[31:0]	代替データ値

cu. po_cdp

表 1.100 po_cdp レジスタフィールド

フィールド (ビット)	名称	説明
31:0	cdp[31:0]	現データのアドレス

cv. po_len

表 1.101 po_len レジスタフィールド

フィールド (ビット)	名称	説明
31:0	len[31:0]	オペランド長

cw. po_said

表 1.102 po_said レジスタフィールド

フィールド (ビット)	名称	説明
31:0	ad	オペランドデータのコプロセッサアドレス (又はデータ)

cx. po_idr

表 1.103 po_idr レジスタフィールド

フィールド (ビット)	名称	説明
31:0	width[31:16] height[15:0]	現画像の画素毎に width を-1する 現画像のライン毎に height を-1する

cy. po_muv_valid

表 1.104 po_muv_valid レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	valid	MUV ラインの有効ビット

cz. po_muv

表 1.105 po_muv レジスタフィールド

フィールド (ビット)	名称	説明
[31:11]	mu_v_base_addr	MUV RAM のベースアドレス

主データパスレジスタ

トされる。

da. mdp_cfg すべてのビットは0にリセットされる。

表 1.106 mdp_cfg レジスタフィールド

フィールド (ビット)	名称	説明
1	E	0=ディスエーブル 1=イネーブル
2	B	0=バイパスしない 1=バイパスする
3	T	0=自己テストモードでない 1=自己テストモード
14	decode	0=命令デコードディスエーブル 1=命令デコードイネーブル mdp_cfg (自己クリア) からセットアップされた mdp_op1, mdp_op2
15	operate	コプロセッサビット (自己クリア)
17	word_input	0=バイト入力に水平補間 1=ワード入力に水平補間
18	hng_kml	カーネル記述子のフォーマット: 0=短いフォーマット 1=長いフォーマット
19	bbnd_gen	0=混合生成要求なし 1=混合生成要求
31:20 (12 bits)	instruction [15:0]	命令の主要、マイナーオペコード

d b. mdp_stat すべてのビットは0にリセットされる。

表 1.107 mdp_stat レジスタフィールド

フィールド (ビット)	名称	説明
0	po_valid	po_valid 信号の状態
1	po_final	po_final 信号の状態
2	po_stall	po_stall 信号の状態
3	oob_valid	oob_valid 信号の状態
4	oob_req	oob_req 信号の状態
5	oob_pending	0=OOB インターフェースに要求なし 1=OOB インターフェースに要求あり
6	ooc_valid	ooc_valid 信号の状態
7	ooc_req	ooc_req 信号の状態

8	ooc_pending	0=00C インターフェースに要求なし 1=00C インターフェースに要求あり
9	ro_valid	ro_valid 信号の状態
10	ro_final	ro_final 信号の状態
11	ro_stall	ro_stall 信号の状態
12:13	matmul_stae[1:0]	行列乗算状態: 0=アイドル 1=行列の左半分乗算 2=行列の右半分乗算
14:15	int_stat[1:0]	固定比例状態での補間: 0=サイクル 0 1=サイクル 1 2=サイクル 2 3=サイクル 3
16	jump	ランプ発生モード (混合モード) 0=ステップモード 1=ジャンプモード
17:18	addgen_state[1:0]	アドレス生成器の状態 0=アイドル 1= PO からオペランドフェッチ 2=アドレス発生
19:20	ramp_state	ランプ発生状態器の状況 0=アイドル 1=動作モードの決定 2=混合発生

d c. mdp_err_int

すべてのビットは0にリセットされる。

表 1.108 mdp_err_int レジスタフィールド

フィールド (ビット)	名称	説明
7:0	error[7:0]	error[0]=ディスエーブル或はMDP がコプロセッサにより操作されていないときに PO からデータ受信 error[1]=ディスエーブル或はMDP がコプロセッサにより操作されていないときに OOB からデータ受信 error[2]=ディスエーブル或はMDP がコプロセッサにより操作されていないときに OOC からデータ受信
24:8	exception[15:0]	exception[0]=チャンネル0 のクランプでのアンダーフロー或はオーバーフロー exception[1]=チャンネル1 のクランプでのアンダーフロー或はオーバーフロー exception[2]=チャンネル2 のクランプでのアンダーフロー或はオーバーフロー exception[3]=チャンネル3 のクランプでのアンダーフロー或はオーバーフロー exception[4]=x 座標のアンダーフロー (画像変換と畳み込み) exception[5]=y 座標のアンダーフロー (画像変換と畳み込み)

dd. mdp_err_int_en

すべてのビットは0にリセットされる。

表 1.109 mdp_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
----------------	----	----

7:0	err_mask[7:0] 1	<p>マスクエラー条件</p> <p>0=マスク</p> <p>1=マスクなし</p> <p>err_mask[0]=ディスエーブル或はMDP がコプロセッサにより操作されていないときに PO からデータ受信</p> <p>err_mask[1]=ディスエーブル或はMDP がコプロセッサにより操作されていないときに OOB からデータ受信</p> <p>err_mask[2]=ディスエーブル或はMDP がコプロセッサにより操作されていないときに OOC からデータ受信</p>
24:8	exp_mask[15:0] 0	<p>マスク例外</p> <p>0=マスク</p> <p>1=マスクなし</p> <p>exp_mask[0]=チャンネル0 のクランプでのアンダーフロー或はオーバフロー</p> <p>exp_mask[1]=チャンネル1 のクランプでのアンダーフロー或はオーバフロー</p> <p>exp_mask[2]=チャンネル2 のクランプでのアンダーフロー或はオーバフロー</p> <p>exp_mask[3]=チャンネル3 のクランプでのアンダーフロー或はオーバフロー</p> <p>exp_mask[4]=x 座標のアンダーフロー (画像変換と畳み込み)</p> <p>exp_mask[5]=y 座標のアンダーフロー (画像変換と畳み込み)</p>

de. mdp_test すべてのビットは0にリセットされる。

表 1.110 mdp_test レジスタフィールド

フィールド (ビット)	名称	説明
3:0	test_data[3 : 0]	テストモードでの RO へのデータ源 0-3=PO データからの RO データ (0,1,2,3 のセッ トは同じ効果) 4=oob_data [31:0] からの RO データ 5=oob_data [63:32] からの RO データ 6=oob_data [95:64] からの RO データ 7=oob_data [127:96] からの RO データ 8=oc_data [31:0] からの RO データ 9=oc_data [63:32] からの RO データ 10=oc_data [95:64] からの RO データ 11=oc_data [127:96] からの RO データ 12-15=reserved
25:4	reserved	
29:26	output_delay[3:0]	入力と出力-2 との間のクロックサイクル数
31:30	skeletal_mode [1:0]	スケレタル MDP モード 0=いくつかの遅延で PO→RO 1=いくつかの遅延で PO→OOB→OC→RO 2=PO, OOB, OC からのデータの最下位バイトが組み 合わされて、結果のワードとして RO に送られる 3=reserved

df mdp_op1 すべてのビットは 0 にリセット される。

表 1.111 mdp_op1 レジスタフィールド

フィールド (ビット)	名称	説明
----------------	----	----

2 : 0	ppb_modeA[2:0]	前処理ブロックの多重機能ブロック A のモード
5 : 3	ppb_modeB[2:0]	前処理ブロックの多重機能ブロック B のモード
8 : 6	pba_modeA[2:0]	ステージ A 処理ブロックの多重機能ブロック A のモード
11 : 9	pba_modeB[2:0]	ステージ A 処理ブロックの多重機能ブロック B のモード
14 : 12	pba_modeC[2:0]	ステージ A 処理ブロックの多重機能ブロック C のモード
17 : 15	pbb_modeA[2:0]	ステージ B 処理ブロックの多重機能ブロック A のモード
20 : 18	pbb_modeB[2:0]	ステージ B 処理ブロックの多重機能ブロック B のモード
23 : 21	pbb_modeC[2:0]	ステージ B 処理ブロックの多重機能ブロック C のモード

27:24	inst_type[3:0]	命令タイプ 0=何もしない(MDP がその命令に対して何もしない) 1=GCSC 2=reserved 3=行列乗算 4=コンボリεύジョン 5=画像変換 6=reserved 7=階調 8=残りのマーシナ 9=バイパス (PO から RO へのデータ通過時に MDP が必要としたとき) 10=水平補間 11=垂直補間 12~13=合成 14~15=reserved
28	it_int	画像変換で要求された補間 0=補間なし 1=補間
29	it_acc	画像変換で要求された累積 0=累積なし 1=累積
30	comp_unpre	合成で要求された非前乗算 0=なし 1=非前乗算(un-pre-multiplication)
31	comp_blend	合成で要求された混合発生 0=なし 1=混合発生

d g m d p _ o p 2 すべてのビットは0にリセット される。

表 1.112 mdp_op2 レジスタフィールド

フィールド (ビット)	名称	説明
1:0	mul_A[1:0]	オペランドA 画素の被乗数 0=0 1=1 2=不透明度B 3=不透明度B
3:2	mul_B[1:0]	オペランドB 画素の被乗数 0=0 1=1 2=不透明度B 3=不透明度B
4	reverse	0=合成時にオペランドを反転しない 1=合成時にオペランド反転
5	addgen_mode	アドレス発生モード 0=画像変換モード 1=畳み込みモード
6	self_cfg	0=長いカーネル記述子 1=短いカーネル記述子、他のパラメータは自動生成
7	reserved	
8	ag_the coprocessor	アドレス生成コプロセッサビット。このアドレス生成が終了するとクリアされる

9	bg_the coprocessort	混合合成コプロセッサビット。混合生成が終了すると、クリアされる
10	mat_the coprocessort	コプロセッサビットの行列乗算。コプロセッサの MDPII 内の行列乗算ステートマシーン。po_final 信号によりステートマシーンでクリアされる。(行列乗算命令に対してのみ有効)
11	int_the coprocessort	コプロセッサビットの補間。コプロセッサの MDPII 内の補間ステートマシーン。po_final 信号によりステートマシーンでクリアされる。(水平及び補間命令に対してのみ有効)
12	int_size	0=パックされたバイトの補間 1=パックされていないバイトの或は画素の補間 (垂直及び水平補間でのみ有効)
13	int_4	0=水平補間 1=4ごとの補間垂直及び水平補間でのみ有効)
14	int_vertical	0=水平補間 1=垂直補間
15	lock_step	0=PO, 00B と 00C インターフェースが互いに独立に動作 1=PO, 00B と 00C インターフェースが互いに連結して動作し、データを一緒に受取る
19:16	reserved	

21 : 20	cw_config[1: 0]	クランプ或はラップ構成 0=ラップされた或は非絶対値 1=ラップされた或は絶対値 2=クランプ (オーバーフロー 0xFF、アンダー フロー0x00) 、非絶対値 3=絶対値及びクランプ
22	fr_en	端数丸め構成 0=ディisable (0 を返す) 1 =イネーブル
24 : 23	oob_mode[1:0]	OoB インターフェース動作モード 0=ディisable 1=順次 2=ランダム 3=混合生成
26 : 25	ooc_mode[1:0]	OoC インターフェース動作モード 0=ディisable 1=順次 2=GCSC 3=画素
30:27	trans[3:0]	合成動作 0=このチャンネルからオフセットを引かない 1=このチャンネルからオフセットを引く カラー空間の変換動作 0=このチャンネルの出力値に変換及びクランプを しない 1=このチャンネルの出力値に変換及びクランプを 使用する 画像変換或は畳み込み処理 0=このチャンネルのアキュムレータを0で初期化 1=このチャンネルのアキュムレータを mdp por:0000 で初期化
31		reserved

dh mdp_por すべてのビットは0にリセット される。

表 1.113 mdp_por レジスタフィールド

フィールド (ビット)	名称	説明
[7:0]	offset0[7:0]	チャンネル0の正オペレータのオフセット
[15:8]	offset1[7:0]	チャンネル1の正オペレータのオフセット
[13:16]	offset2[7:0]	チャンネル2の正オペレータのオフセット
[31:24]	offset3[7:0]	チャンネル3の正オペレータのオフセット
[31:0]	offset[31:0]	畳込み(convolution)と変換のオフセット

d i mdp_bi すべてのビットは0にリセットされ、`mdp_bi`レジスタは種々のモードの様々なものに用いられる。

表 1.114 mdp_bi レジスタフィールド (合成モード)

フィールド (ビット)	名称	説明
[7:0]	blendend0	チャンネル0の混合のコプロセッサ値
[15:8]	blendend1	チャンネル1の混合のコプロセッサ値
[23:16]	blendend2	チャンネル2の混合のコプロセッサ値
[31:24]	blendend3	チャンネル3の混合のコプロセッサ値

表 1.115 mdp_bi レジスタフィールド (非合成モード)

フィールド (ビット)	名称	説明
[31:2]	ioffset	インデックステーブルのオフセット

d j mdp_bm すべてのビットは0にリセットされ、`mdp_bm`レジスタは異なるモードの異なるものに用いられる。

表 1.116 mdp_bm レジスタフィールド (合成モード)

フィールド (ビット)	名称	説明
[7:0]	blendend0	チャンネル0の混合の最終値
[15:8]	blendend1	チャンネル1の混合の最終値
[23:16]	blendend2	チャンネル2の混合の最終値
[31:24]	blendend3	チャンネル3の混合の最終値

表 1.117 mdp_bm レジスタフィールド (非合成モード)

フィールド (ビット)	名称	説明
[3:0]	rows[3:0]	行列の行(row)の数
[7:4]	cols[3:0]	行列の列(column)の数
[15:8]	level[7:0]	階調のレベル数
[20:16]	bp[4:0]	2進数ポイントの位置

d k mdp_len すべてのビットは0にリセットされ、`mdp_len`レジスタは異なるモードの異なるものに用いられる。

表 1.118 mdp_len レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	length	生成されるべき混合長

JPEG符号化器レジスタ dl jc_cfg

表 1.119 JPEG命令用 jc_cfg レジスタフィールド

フィールド (ビット)	名称	説明
[31:28]	mop	命令=0010からの主オペコード
[27]	D	0=JPEG 圧縮 1=JPEG 伸長
[26]	M	0=単一カラーチャネル 1=多重カラーチャネル
[25]	4	0=3チャネル 1=4チャネル
[24]	S	0=サブサンプリングレジメを使用しない 1=サブサンプリングレジメを使用する
[23]		reserved
[22]	H	0=高速ハフマンアルゴリズムを使用 1=低速ハフマンアルゴリズムを使用
[21:16]		reserved
[15]	O	0=JC が非動作 1=JC が動作
[14]	dec	0=命令のデコードディスエーブル 1=命令のデコードイネーブル
[13:8]		reserved
[7]	A	0=命令のデコードディスエーブル 1=命令のデコードイネーブル

[6]	Z	0=1のパッド 1=0のパッド
[5:4]		reserved
[3]	T	0=自己テストモードでない 1=自己テストモード
[2]	B	0=バイパスしない 1=バイパスする
[1]	E	0=ディスエーブル 1=イネーブル
[0]		reserved

表 1.120 データ符号化用 jc cfg レジスタフィールド

フィールド (ビット)	名称	説明
[31:28]	mop	命令=0010 からの主オペコード
[27]	D	0=圧縮 1=伸長
[26]		reserved
[25:24]	diff	入力バイトと出力バイト数の差 00=差がない 01=入力バイト数よりも出力バイト数が多い 10=11=出力バイト数よりも入力バイト数が少ない
[23]	op	動作 0=ハフマン 1=バイ符号化
[22:19]		reserved
[18:16]	lbo	入力ビットオフセット

[15]	0	0=JC が非動作 1=JC が動作
[14]	dec	0=命令のデコードディスエーブル 1=命令のデコードイネーブル
[13:7]		reserved
[6]	Z	0=1のパッド 1=0のパッド
[5:4]		reserved
[3]	T	0=自己テストモードでない 1=自己テストモード
[2]	B	0=バイパスしない 1=バイパスする
[1]	E	0=ディスエーブル 1=イネーブル
[0]		reserved

表 1.121 メモリコピー用命令 jc cfg レジスタフィールド

フィールド (ビット)	名称	説明
[31:28]	mop	命令=1001 からの主オペコード
[27]	D	0=一般的なデータ移動 1=ローカルDMA このビットは常に0にセットされる
[26]	B	0=ビットコピー動作でない 1=ビットコピー動作

[25:24]	diff	入力バイトと出力バイト数の差 00=差がない 01=入力バイト数よりも出力バイト数が多い 10=11=出力バイト数よりも入力バイト数が少ない
[23]		reserved
[22:20]	obo	出力ビットオフセット
[19]		reserved
[18:16]	ibo	入力ビットオフセット
[15]	o	0=JC が非動作 1=JC が動作
[14]	dec	0=命令のデコードディスエーブル 1=命令のデコードイネーブル
[13:4]		reserved
[3]	T	0=自己テストモードでない 1=自己テストモード
[2]	B	0=バイパスしない 1=バイパスする
[1]	E	0=ディスエーブル 1=イネーブル
[0]		reserved

dm jc_stat

dn jc_err_int

表 1.122 jc_stat レジスタフィールド

フィールド (ビット)	名称	説明
[31:8]	reserved	
[7:0]	state	最終化

表 1.123 jc_err_int レジスタフィールド

フィールド (ビット)	名称	説明
[31:19]		reserved
[18]	huff_ill_tables	不法なハフマンテーブル。9以上のハフマン テーブルのミスが発生
[17]	huff_ill_mpos	不法なマーカ位置
[16]	huff_ill_marker	不法なマーカ
[15]	coeff_ill_AC	不法なAC係数値(-1024)
[14]	coeff_ill_DC	不法なDC値
[13]	coeff_ill_AC_mag	不法なACの大きさカテゴリ
[12]	coeff_ill_DC_mag	不法なDCの大きさのカテゴリ
[11]	coeff_ill_RST	不法なRSTmのカウント値
[10]	coeff_ill_overflow	係数デコーダにより検知されたデータのオー バーフロー
[9]	coeff_ill_mpos	不法なマーカ位置
[8]	jpeg_unerflow	アンダーフロー
[1]	jpeg_disable	ディスエーブルエラーのときにデータを受信
[0]	huff_ill_symbol	不法なハフマン・シンボル・エラー

do jc_err_int_en

表 1.124 jc_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
[31:19]		reserved
[18]	huff_ill_tables	不法なハフマンテーブル。9以上のハフマン テーブルのミスが発生
[17]	huff_ill_mpos	不法なマーカ位置

[16]	huff_ill_marker	不法なマーカ
[15]	coeff_ill_AC	不法なAC係数値(-1024)
[14]	coeff_ill_DC	不法なDC値
[13]	coeff_ill_AC_mag	不法なACの大きさカテゴリ
[12]	coeff_ill_DC_mag	不法なDCの大きさのカテゴリ
[11]	coeff_ill_RST	不法なRSTmのカウント値
[10]	coeff_ill_overflow	係数デコーダにより検知されたデータのオーバーフロー
[9]	coeff_ill_mpos	不法なマーカ位置
[8]	jpeg_underflow	アンダーフロー
[1]	jpeg_disable	ディスエーブルエラーのときにデータを受信
[0]	huff_ill_symbol	不法なハフマン・シンボル・エラー

dp jc_rsi

表 1.125 jc_rsi レジスタフィールド

フィールド (ビット)	名称	説明
[15:0]	rsi	コプロセッサマーカの間のMCUブロックの数

dq jc_decode

表 1.126 jc_decode レジスタフィールド

フィールド (ビット)	名称	説明
[0]	dct_eneb	サブモジュールのイネーブル
[1]	dct_bypass	dctサブモジュールをバイパスモードにする
[2]	dct_forward	dctサブモジュールを順モードにする
[3]	qdq_eneb	量子化器サブモジュールをイネーブル
[4]	qdq_bypass	量子化器サブモジュールをバイパス
[5]	qdq_forward	量子化器サブモジュールをバイパス

[6]	qdq_four	4チャンネル画像
[7]	qdq_subsmpl	サブサンプルされた画像
[8]	cc_enable	係数符号器サブモジュールをイネーブルにする
[9]	cc_bypass	係数符号器サブモジュールをバイパスする
[10]	cc_forward	係数符号器を順方向モードにする
[11]	cc_peg	JPEG 準拠ストリームの符号化
[12]	cc_subsmpl	
[13]	cc_fourchannel	
[14]	ccm_u1channel	
[15]	hc_enable	ハフマン符号化器サブモジュールをイネーブル
[16]	hc_bypass	ハフマン符号化器をバイパス
[17]	hc_forward	ハフマン符号化器を順方向モードにする
[18]	hc_subsmpl	サブサンプルされた画像
[19]	hc_fast	高速ハフマン符号化
[20]	hc_peg	JPEG 準拠のハフマン符号化実行
[21]	hc_four	4チャンネル画像
[22]	hc_align	RSTm マーカをワードの境界に割り当てる
[23]	hc_zeropad	0=1を使用したパッド 1=0を使用したパッド
[24]	hc_memory	メモリコピー操作実行
[25]	misc_forward	
[26]	qdp_multi	
[27]	hc_multi	
[31:22]		reserved

dr jc_res

ds jc_table_sel

表 1.127 jc_res レジスタフィールド

フィールド (ビット)	名称	説明
[7:0]	res	残りの値

表 1.128 jc_table_sel レジスタフィールド

フィールド (ビット)	名称	説明
[13:8]	j_table_sel_quant	量子化のための復号テーブル選択
[5:0]	j_table_sel_huff	ハフマン復号のための復号テーブル 選択

結果オーガナイザレジスタ

dt ro_cfg

表 1.129 ro_cfg レジスタフィールド

フィールド (ビット)	名称	説明
[31:23]	reserved	
[22:21]	chan	JPEG 伸長出力データフォーマット: 0,1=1つのチャネル 2=3チャネル 3=4チャネル
[20]	upsample	0=アップサンプリングしない 1=MUV RAM からのアップサンプリングデータ
[19:18]	reserved	
[17]	use_out	0=出力データをカットするのにカットレジスタ値を使用しない 1=出力データをカットするのにカットレジスタ値を使用。このビットは命令の完了時に0に戻される

[16]	use_limit	0=出力データを限定するのに限定レジスタ値を使用しない 1=出力データを限定するのに限定レジスタ値を使用。このビットは命令の完了時に0に戻される
[15:14]		reserved
[13:12]	mode[1:0]	現在の動作モード: 0=アイドル 1=順次 2=JPEG 伸長 3=Cbus データ
[11:4]		reserved
[8]	S	0=自己テストモードでない 1=自己テストモード
[2]	B	0=バイパスしない 1=バイパスする
[1]	E	0=ディスエーブル 1=イネーブル
[0]		reserved

du ro_stat

表 1.130 ro_stat レジスタフィールド

フィールド (ビット)	名称	説明
[31:24]		reserved
[23:16]	state [7:0]	内部状態条件: ビット0=R0 がストール ビット1=R0 がアクティブ ビット2=FF0 が一杯 ビット3=FF0 が空
[15:0]		reserved

dv ro_err_int

表 1.131 ro_err_int レジスタフィールド

フィールド (ビット)	名称	説明
[7:0]	error	エラー条件: ビット0=ディスエーブル時に要求 ビット1=MDP と JPEG 符号化器が同時アクティ ブ ビット2=不正なアドレスエラー ビット3=JPEG モードでのデータオーバフロー

dw ro_err_int_en

表 1.132 ro_err_int_en レジスタフィールド

フィールド (ビット)	名称	説明
[7:0]	error_mask	エラー条件マスク。対応するエラーをイネ ーブルにする

dx ro_dmr

表 1.133 ro_dmr レジスタフィールド

フィールド (ビット)	名称	説明
[31:30]	b3	バイト3のレインスワップ: 0=バイト0から 1=バイト1から 2=バイト2から 3=スワップなし

[29:28]	b2	<p>バイト2のレインスワップ:</p> <p>0=バイト0から</p> <p>1=バイト1から</p> <p>2=スワップなし</p> <p>3=バイト3から</p>
[27:26]	b1	<p>バイト1のレインスワップ:</p> <p>0=バイト0から</p> <p>1=スワップなし</p> <p>2=バイト2から</p> <p>3=バイト3から</p>
[25:24]	b0	<p>バイト0のレインスワップ:</p> <p>0=スワップなし</p> <p>1=バイト1から</p> <p>2=バイト2から</p> <p>3=バイト3から</p>
[23:20]	suben[3:0]	<p>入替えイネーブル:</p> <p>0=このバイトの入替えを実行しない</p> <p>1=このバイトにro_substに記憶された値を使用</p>
[19:16]	wrmask	<p>書き込みマスク:</p> <p>0=対応するバイトチャネルに書き込む</p> <p>1=対応するバイトチャネルに書き込まない</p>
[15]	cmbs	<p>最上位ビットの選択:</p> <p>0=逆正規化のときにバイトの最下位ビットを選択</p> <p>1=逆正規化のときにバイトの最上位ビットを選択</p>

[4:12]	nom a lse	逆正規化(denorm alization)要因: 0=1ビットデータ 1=2ビットデータ 2=4ビットデータ 3=8ビットデータ 4~7=16ビットデータ対象
[18]		reserved
[7]	P	外部フォーマット: 0=パックされていないバイト 1=パックされたバイト
[6:5]	f	内部フォーマット: 0=直読 1=パックされていないバイト 2=パックされたバイト 3=その他
[4:3]	cc	チャネルカウント: 0=4アクティブチャネル 1=1アクティブチャネル 2=2アクティブチャネル 3=3アクティブチャネル
[2:0]		reserved

dy ro_subst

表 1.134 ro_subst レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	subst[31:0]	Cbus モードの代替値またはデータ値

dz ro_cdp

表 1.135 ro_cdp レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	cdp[31:0]	現データのアドレス

ea ro_len

表 1.136 ro_len レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	ln[31:0]	出力バイト数カウント

eb ro_sa

表 1.137 ro_sa レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	sa[31:0]	コプロセッサのアドレス

ec ro_idr

表 1.138 ro_idr レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	height[5:0] width[31:16]	height-1、現画像のラインにおける width-1、現画像の画素における

ed ro_vbase

表 1.139 ro_vbase レジスタフィールド

フィールド (ビット)	名称	説明
[31:0]	vbase[31:12]	コプロセッサの仮想アドレスベース

ee ro_cut

表 1.140 ro_cut レジスタフィールド

フィールド (ビット)	名称	説明
[31:12]	cut[31:0]	出力カットオフセット、多くのバイトをずてる

ef ro_lmt

表 1.141 ro_lmt レジスタフィールド

フィールド (ビット)	名称	説明
hlt[31:0]	hlt[31:0]	出力バイト数のリミット

PCI コンフィギュレーション空間のエイリアス PCI コンフィギュレーション空間は 256 バイトの、PCI によって定義されたレジスタのブロックであり、ホストが PCI デバイスをコンフィギュレーションしたり、その状態を読んだりすることを認めている。それは PCI コンフィギュレーションサイクルを用いてアクセスされる。レジスタはまたコプロセッサの内部メモリの読み込み専用エリアにミラーされており、従って PCI の通常のメモリサイクルを用いて読むことができる。EIC に実装されているコンフィギュレーション空間のフォーマットを表 1.141.1 に示す。

表 1.141.1 コプロセッサ PCI 構成の空間的レイアウト

表 1.141.1

31		1615		0
デバイス ID		ベンダー ID		
ステータス		コマンド		
クラスコード			レビジョン ID	
Reserved	ヘッダタイプ	タイマ	キャッシュ ラインサイズ	

ベースアドレス			
Reserved			
システム ID		サブシステムベンダー ID	
Reserved			
Max_Lat	Min_Gnt	Interrupt_Pn	Interrupt Line

予約のレジスタと実装されたレジスタにおける予約のビットは読み込みに対しては0を返し、また書き込みによって影響しない。0x40-0xffの範囲のコンフィギュレーション空間のアドレスもまた予約である。ベンダー専用のコンフィギュレーションレジスタは定義されない。

e g ベンダーID

このレジスタは読み込み専用である。CISRAのベンダーIDは0x11ACである。

e h デバイスID

このレジスタは読み込み専用である。コプロセッサのデバイスIDは0x0001である。デバイスIDフィールドは二つの8ビットのフィールドに分割されている：最上位の8ビットはデバイスの特徴を示す番号（0x0はコプロセッサ）で、最下位の8ビットはそのデバイスのバージョン番号（0x1はコプロセッサのバージョン）を示す。

e i コマンドレジスタ

コマンドレジスタのフィールドの定義を表1.142に示す。このレジスタのすべての予約されていないビットは読みこみ／書き込みができる。リセット後にはこのレジスタは0x0000にセットされる。

表 1.142 コプロセッサPCIコマンドレジスタフィールド

フィールド (ビット)	名称	説明
0	IO スペース	reserved

1	Memory Space	コプロセッサがメモリサイクルに応答可能
2	Bus Master	コプロセッサが PCI バス上でサイクルの生成を可能
3	Special Cycles	reserved
4	Memory Write & Invalidate Enable	reserved
5	VGA Pallete Snoop	reserved
6	Parity Error Response	データパリティエラー時に perr_1 を出力し、アドレスパリティエラー時に serr_1 を出力可能。また、ステータスレジスタにパリティエラー検知ビットをセット
7	Wait sytle control	reserved
8	serr_L Enable	serr_1 の出力可能。コプロセッサはアドレスパリティエラーを検知したときのみ serr_1 を出力
9	Fast back-to-back Enabel	reserved
10-15	reserved	reserved

e j ステータスレジスタ ステータスレジスタのフィールドの定義を表 1. 1 4 3 に示す。このレジスタの読み込みは通常通りである。このレジスタのいくつかのビットは読み込み専用である。その他のビットはコプロセッサのみにより 1 にセットされ、ホストのみによって 0

にリセットされる（テストモードを除く）。ホストはそのビットに 1 を書き込むことでリセットする；0 の書き込みは意味をなさない。リセット後にはこのレジスタは 0 x 0 2 8 0 にセットされる。

表 1.143 コプロセッサPCIステータスレジスタフィールド

フィールド (ビット)	名称	説明
0-4	reserved	reserved
5	66MHz capable	reserved
6	User Definable Features	reserved
7	Fast back- to-back Capable	このビットは読出し専用で1にセットされる。これはコプロセッサが、高速バック・ツー・バックサイクルが別のターゲットデバイスにアドレスされたシステムで現在動作していることを示す
8	Data Parity Error Detected	コプロセッサがマスタとしてリードデータのパリティエラーを検知したとき、或は書込み時に perr_1 を検知したときにセットされる。コマンドレジスタのビット6がセットされているときのみセットされ、このビットがセットされると割り込みが発生する

10-9	devsel_1 timing	読出し専用ビットで 01 にセットされ、コ プロセッサがターゲットの時、中速で deversel_1 を出力して応答しているこ とを示す
11	Signalled Target Abort	reserved
12	Received Target Abort	マスタであるコプロセッサがターゲット の失敗 (abort) を受けるとセットされる 。これにより EIC が割り込みを発生し、 このビットがソフトウェアによりクリア される迄動作を停止する
13	Received Master Abort	マスタであるコプロセッサがマスタの失 敗 (abort) を実行するとセットされる。 これにより EIC が割り込みを発生し、こ のビットがソフトウェアによりクリアさ れる迄動作を停止する
14	Signalled System Error	コプロセッサが serr_1 を出力するとセ ットされる。これはアドレスパリティエ ラーの時のみ発生する
15	Detected Parity Error	コプロセッサがアドレス或はデータパ リティエラーを検知するとセットされる

e k リビジョン ID これは読み込み専用のレジスタである。コプロセッサの初期リビジョン ID は 0 x 01 である。 e l クラスコード これは読み込み専用のレジスタである。コプロセッサは PC I S I G の定義されたクラスコードに適さないこのレジスタは 0 x F F 0 0 0 0 にセットされる。

e m キャッシュラインサイズ

これは 3 2 ビットワード単位でシステムのキャッシュラインサイズを決定する読み書き可能なレジスタである。これはコプロセッサがメモリ読み込みラインやメモリ多重読み込みコマンドを使用するときに決定する。コプロセッサはこのレジスタの 0 から 2 5 5 までの値をサポートする。このレジスタにおける 0 はメモリ読み込みラインおよびメモリ多重読み込みの形式を無効にする。このレジスタはリセット時には 0 x 0 0 にセットされる。

e n 遅延タイマ

このレジスタはすべての PC I の処理に CPU が使用する最大のクロック数を特定する読み書きできるレジスタである。コプロセッサはこのレジスタにおいて 0 から 2 5 5 の値をサポートする。このレジスタはリセット時には 0 x 0 0 にセットされる。

e o ヘッダタイプ

この読み込み専用のレジスタは 0 x 0 0 にセットされる。このことはコプロセッサがタイプ 0 のレイアウトのコンフィギュレーション空間を使用することを意味する。

e p ベースアドレス

この読み書き可能なレジスタはコプロセッサの内部レジスタ、内部メモリ、ローカルメモリ、及び汎用インターフェースをホストのメモリマップ内に配置するために用いられる。コプロセッサの様々なリソースは 6 4 MB (すべてが使用される訳ではない) を占有し、従ってこのレジスタの先頭 6 ビットだけが書き込み可能である。残りのビットはすべて 0 にハード的に結線されている。このレジスタの下位の 4 ビットは読み込み専用の制御ビットであり、これらもまた 0 に結線されている。このことはレジスタがメモリ空間を参照することを意味し、コプロセッサがホスト側の 3 2 ビット空間のどこにでもマッピングされ、コプロセッサのリソースがターゲットであるときはプリフェッチできないことを意味する。

e q サブシステムベンダー ID

この読み込み専用レジスタはホストがシステムに実装さ

れたPCIボードのベンダーを識別できるようにする
(ボード上のPCIインターフェースに実装したコンポーネントのベンダーに対して)。このレジスタの内容はリセット時にEICコンフィギュレーションシリアルポートを用いてロードされる。

er サブシステムID

この読み込み専用レジスタはホストがシステムに実装されたPCIボードを識別できるようにする。このレジスタの内容はリセット時にEICコンフィギュレーションシリアルポートを用いてロードされる。このメカニズムはボードの機能あるいはコンフィギュレーションに必要な情報の外部からの符号化およびホストからの読み込みを可能にする。

es 割り込みライン

この読み書きできるレジスタはシステムソフトウェアが割り込みラインルーティング情報を記録できる様にするために使用され、割り込みサービスソフトウェアによりアクセスできる。コプロセッサ内の処理には全く影響を与えない。このレジスタはリセット時には0x00にセ

ットされる。

et 割り込みピン

この読み込み専用レジスタはハード的に0x01に結線されている。このことはコプロセッサがPCIのinta_1割り込みピンを使用することを示す。

eu Min_Gnt

この読み込み専用レジスタはコプロセッサが要求する1/4マイクロ秒単位のバースト期間長をホストに示す。このレジスタの最適な値はまだ決まっていない。

ev Max_Lat

この読み込み専用レジスタは1/4マイクロ秒単位での、コプロセッサが要求するPCIバスのゲインコントロール最大遅延をホストに示す。このレジスタの最適な値はまだ決まっていない。

1. 1. 4 内部メモリマップ

本節ではコプロセッサの内部メモリマップ内のプレモジュールデータエリアに生ずるオブジェクトの詳細について述べる。

表 1.144 コプロセッサ内部メモリマップ

コプロセッサ のバスから のバイトオフ セット	関連するモ ジュール	名称	タイプ	定義
0x0000- 0x1FFFF	reserved	reserved		
0x8000- 0x803F	EIC	eic_ptp	R/W	MMU のページテーブルポ インタ
0x8040- 0x07F	reserved	reserved		
0x8080- 0x80FF	EIC	eic_addr	Read Only	仮想及び物理ページ番号 を持つ CAM データがイン ターレースされる
0x8100- 0xFFFF	reserved	reserved		
0x10000- 0x1FFFF	LMC	reserved		
0x20000- 0x2007F	PIC	input RAM	R/W	PTC 入力データバスの 32 ワード×32 ビット RAM
0x20080- 0x200FF	PIC	output RAM	R/W	PTC 出力データバスの 32 ワード×32 ビット RAM
0x20100- 0x2FFFF	PIC	reserved		
0x30000- 0x3FFFF	MISC	reserved		
0x40000- 40017	IC	prefetch _buf	read only	先取りバッファの内容

0x40018- 0x4FFFF	LC	reserved		
0x50000	DCC	cache RAM	R/W	32ビットで 4k バイト のワイドメモリ
0x51000	DCC	tag RAM	R/W	128×20 ビットのタグメ モリ。 32ビットワードの上位 20ビットが使用される
0x52000	DCC	dcc_odat ab0	R/W	o_datab バスのビット 31:0。テストモードでバ スに出力される
0x52004	DCC	dcc_odat ab1	R/W	o_datab バスのビット 63:32。テストモードでバ スに出力される
0x52008	DCC	dcc odatab2	R/W	o_datab バスのビット 95:64。テストモードでバ スに出力される
0x5200c	DCC	dcc_odat ab3	R/W	o_datab バスのビット 127:96。テストモードで バスに出力される
0x52010	DCC	dcc_odat ac0	R/W	o_datac バスのビット 31:0。テストモードでバ スに出力される
0x52014	DCC	dcc_odat ac1	R/W	o_datac バスのビット 63:32。テストモードでバ スに出力される

0x52018	DCC	dcc odatac2	R/W	o_datac バスのビット 95:64。テストモードでバ スに出力される
---------	-----	----------------	-----	---

コプロセッサ のバスから のバイトオフ セット	関連するモ ジュール	名称	タイプ	定義
0x5201c	DCC	dcc_odata c3	R/W	o_datac バスのビット 127:96。テストモードでバ スに出力される
0x060000- 0x0607FF	P0	mu_v_ram	R/W	Muv RAM. MUV RAMは、 内部及び Fraction RAM と 同じフォーマット
0x060800- 0x067FF	P0	reserved		
0x70000- 0x7FFFF	00B	reserved		
0x80000- 0x8FFFF	00C	reserved		
0x90000 - 0x900FC	JC	Quantizer Buffer 1		最下位の15ビットを使用 。上位の17ビットは予約 済み
0x90100- 0x900FC	JC	Quantizer Buffer 2		最下位の15ビットを使用 。上位の17ビットは予約 済み

0x90200- 0x902FC	JC	DCT Buffer		最下位の12ビットを使用 。上位の20ビットは予約 済み
0x90300- 0x9FFFF	JC	reserved		
0x0A0000- 0x0A01FF	MDP			内部構造
0x0A0200- 0x0AFFFF	MDP	reserved		
0x0B0000- 0x0BFFFF	RO	reserved		
0xC0000- 0xCFFFF		reserved		
0xD0000- 0xDFFFF	IIS	reserved		
0xE0000- FFFFFF		reserved		

1. 1. 5 メモリワードフィールド

a e i c _ p t p

表 1.145 e i c _ p t p メモリワードフィールド

フィールド (ビット)	名称	説明
11:0	reserved	
31:12	ptp	ページテーブルの 4K バイトセグメン トの基底の物理バイトアドレスの上か ら 20 ビット

【図面の簡単な説明】

- 【図 1】 ホストコンピュータ環境内のラスト画像コプロセッサの動作を示す図
- 【図 2】 図 1 のラスト画像コプロセッサをより詳細に示した図
- 【図 3】 ラスタ画像コプロセッサのメモリマップを示す図
- 【図 4】 CPU、命令キュー、命令オペランド、共有メモリ中の結果、コプロセッサ間の関係を示す図
- 【図 5】 命令生成部、メモリ管理部、キュー管理部、コプロセッサ間の関係を示す図
- 【図 6】 命令をペンディング命令キューから読み込み、終了命令キューに配置するグラフィックスコプロセッサの動作を示す図
- 【図 7】 命令キューの固定長巡回バッファ実装を示し、バッファが溢れるまで待機する必要性を説明する図
- 【図 8】 コプロセッサにおいて用いられる命令実行ス

トリームを示す図

- 【図 9】 命令実行フローチャート、
- 【図 10】 コプロセッサにおいて用いられる標準命令ワードフォーマットを示す図
- 【図 11】 標準命令の命令ワードフィールドを示す図
- 【図 12】 標準命令のデータワードフィールドを示す図
- 【図 13】 図 2 の命令制御部を模式的に示す図
- 【図 14】 図 13 の実行制御部をより詳細に示した図
- 【図 15】 命令制御部の状態遷移図
- 【図 16】 図 13 の命令復号部を示す図
- 【図 17】 図 16 の命令シーケンサをより詳細に示した図
- 【図 18】 図 16 の ID シーケンサの状態遷移図
- 【図 19】 図 13 のプレフェッチバッファ制御部をより詳細に示した図
- 【図 20】 コプロセッサで用いられるレジスタ記憶とモジュール間関連の標準形式を示す図

【図21】 コプロセッサにおいて用いられる制御バス処理のフォーマットを示す図

【図22】 コプロセッサの一部内のデータフローを示す図

【図23】 コプロセッサにおいて用いられるさまざまなデータ再フォーマット例を示す図

【図24】 コプロセッサにおいて用いられるさまざまなデータ再フォーマット例を示す図

【図25】 コプロセッサにおいて用いられるさまざまなデータ再フォーマット例を示す図

【図26】 コプロセッサにおいて用いられるさまざまなデータ再フォーマット例を示す図

【図27】 コプロセッサにおいて用いられるさまざまなデータ再フォーマット例を示す図

【図28】 コプロセッサにおいて用いられるさまざまなデータ再フォーマット例を示す図

【図29】 コプロセッサにおいて用いられるさまざまなデータ再フォーマット例を示す図

【図30】 コプロセッサにおいて実行されるフォーマット変換を示す図

【図31】 コプロセッサにおいて実行されるフォーマット変換を示す図

【図32】 コプロセッサにおいて実行される入力データ変換処理を示す図

【図33】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図34】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図35】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図36】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図37】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図38】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図39】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図40】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図41】 コプロセッサにおいて実行されるさまざまなデータ変換を示す図

【図42】 コプロセッサにおいて実行されるさまざまな内部から出力データ変換を示す図

【図43】 コプロセッサにおいて実行されるさまざまなデータ変換例を示す図

【図44】 コプロセッサにおいて実行されるさまざまなデータ変換例を示す図

【図45】 コプロセッサにおいて実行されるさまざまなデータ変換例を示す図

【図46】 コプロセッサにおいて実行されるさまざまなデータ変換例を示す図

【図47】 コプロセッサにおいて実行されるさまざまなデータ変換例を示す図

【図48】 どのデータ変換が用いられるべきかを決定する内部レジスタで用いられるさまざまなフィールドを示す図

【図49】 データ正規化を用いるグラフィックスサブシステムのブロック図

【図50】 データ正規化装置の回路図

【図51】 合成処理において実行されるピクセル処理を示す図

【図52】 合成処理のための命令ワードフォーマットを示す図

【図53】 合成処理のためのデータワードフォーマットを示す図

【図54】 タイル処理のための命令ワードフォーマットを示す図

【図55】 画像に対するタイル命令の動作を示す図

【図56】 色値を再マッピングするための色区間／区間内位置テーブルの利用処理を示す図

【図57】 コプロセッサのMUVバッファ内の区間／区間内位置テーブルの格納形式を示す図

【図58】 コプロセッサにおいて実行される補間を用いた色変換処理を示す図

【図59】 コプロセッサにおいて実行されるエッジでの色変換処理の改善処理を示す図

【図60】 コプロセッサにおいて実行される1出力色のための色空間変換処理を示す図

【図61】 単一色出力色空間変換を用いたときのコプロセッサのキャッシュ内でのメモリ格納を示す図

【図62】 複数色空間変換で用いられる手法を示す図

【図63】 複数色空間変換処理において用いられるキャッシュのためのアドレス再マッピング処理を示す図

【図64】 色空間変換命令における命令ワードフォーマットを示す図

【図65】 複数色変換手法を示す図

【図66】 コプロセッサで実行されるJPEG変換処理でのMCUの生成を説明する図

【図67】 コプロセッサで実行されるJPEG変換処理でのMCUの生成を説明する図

【図68】 コプロセッサのJPEG符号化部の構造を示す図

【図69】 図68の量子化部をより詳細に示す図

【図70】 図68のハフマン符号化部をより詳細に示す図

【図71】 ハフマン符号化部と復号部とを示す図

【図72】 ハフマン符号化部と復号部とを示す図

【図73】 コプロセッサで用いられるJPEGデータの削除・制約処理を説明する図

【図74】 コプロセッサで用いられるJ P E Gデータの削除・制約処理を説明する図

【図75】 コプロセッサで用いられるJ P E Gデータの削除・制約処理を説明する図

【図76】 J P E G命令の命令ワードフォーマットを示す図

【図77】 一般の離散コサイン変換装置（従来例）のブロック図

【図78】 従来例のD C T装置の算術データパスを示す図

【図79】 コプロセッサで用いられるD C T装置のブロック図

【図80】 図79の算術回路をより詳細に示すブロック図

【図81】 図79のD C T装置の算術データパスを示す図

【図82】 J P E Gフォーマットのように符号化されていないビットフィールド（バイト整列されているものとされていないもの）がインタリーブされた代表的なハフマン符号化データを示す図

【図83】 図84のJ P E Gデータのハフマン復号部の全体の構造をより詳細に示した図

【図84】 J P E Gデータのハフマン復号部の全体の構造を示す図

【図85】 バイト整列された符号化されていないビットフィールドを入力データから削除するストリップブロック中のデータ処理を示し、ストリップから出力されるデータに対応するタグ符号の例をも示す図

【図86】 データプレシフタの構成とデータフローを示す図

【図87】 図81の復号部の制御ロジックを示す図

【図88】 マーカプレシフタの構成とデータフローを示す図

【図89】 J P E G符号化においてハフマン符号値を復号する組み合わせ回路のブロック図、

【図90】 パディング領域の概念とパディングビットの復号部のブロック図

【図91】 復号部から出力され、コプロセッサにおいて用いられるデータフォーマットの例を示す図

【図92】 画像変換命令において用いられる手法を示す図

【図93】 画像変換命令における命令ワードフォーマットを示す図

【図94】 コプロセッサで用いられる画像変換カーネルのフォーマットを示す図

【図95】 コプロセッサで用いられる画像変換カーネルのフォーマットを示す図

【図96】 コプロセッサで用いられる画像変換のためのインデックステーブルの利用処理を示す図

【図97】 変換や畳込みで用いる命令のためのデータ

フィールドフォーマットを示す図、

【図98】 命令ワードのb pフィールドの説明図

【図99】 コプロセッサで用いられる畳込み処理を示す図

【図100】 コプロセッサで用いられる畳込み命令の命令ワードフォーマット図

【図101】 コプロセッサで用いられる行列乗算の命令ワードフォーマット図、

【図102】 コプロセッサで用いられる階層的画像操作処理を示す図

【図103】 コプロセッサで用いられる階層的画像操作処理を示す図

【図104】 コプロセッサで用いられる階層的画像操作処理を示す図

【図105】 コプロセッサで用いられる階層的画像操作処理を示す図

【図106】 階層的画像命令での命令ワード符号を示す図

【図107】 コプロセッサで用いられるフロー制御命令の命令ワード符号を示す図

【図108】 ピクセルオーガナイザをより詳細に示す図

【図109】 ピクセルオーガナイザにおけるオペランドフェッチ部をより詳細に示す図

【図110】 コプロセッサで用いられる種々の格納フォーマットを示す図

【図111】 コプロセッサで用いられる種々の格納フォーマットを示す図

【図112】 コプロセッサで用いられる種々の格納フォーマットを示す図

【図113】 コプロセッサで用いられる種々の格納フォーマットを示す図

【図114】 コプロセッサで用いられる種々の格納フォーマットを示す図

【図115】 コプロセッサのピクセルオーガナイザにおけるM U Vアドレス生成部をより詳細に示す図

【図116】 コプロセッサで用いられる多重値（M U V）バッファのブロック図

【図117】 図116の符号化器の構造を示す図

【図118】 図116の復号器の構造を示す図

【図119】 J P E Gモード（ピクセル分解）において読み出しアドレスを生成する図116のアドレス生成部の構造を示す図

【図120】 J P E Gモード（ピクセル復元）において読み出しアドレスを生成する図116のアドレス生成部の構造を示す図

【図121】 図116の記憶装置を備えるメモリモジュールの構成を示す図

【図122】 読み出しアドレスをメモリモジュールに多重化する回路の構造を示す図

【図123】 単一ルックアップテーブルモードで動作するバッファ内にルックアップテーブルエントリがどのように格納されるかを示す図

【図124】 多重ルックアップテーブルモードで動作するバッファ内にルックアップテーブルエントリがどのように格納されるかを示す図

【図125】 J P E Gモード（ピクセル分解）で動作するバッファ内にピクセルがどのように格納されるかを示す図

【図126】 J P E Gモード（ピクセル復元）で動作するバッファから単一カラーがどのように格納されるかを示す図

【図127】 コプロセッサの結果オーガナイザの構造をより詳細に示す図

【図128】 コプロセッサのオペランドオーガナイザの構造をより詳細に示す図

【図129】 コプロセッサにおいて用いられる主データパス部のためのコンピュータアーキテクチャのブロック図

【図130】 更なる処理のために入力データオブジェクトを受け取り、格納し、再配列するための入力インターフェースのブロック図

【図131】 入力データオブジェクトに対して算術演算を実行するための画像データプロセッサのブロック図

【図132】 入力データオブジェクトの1つのチャンネルに対して算術演算を実行するためのカラーチャンネルプロセッサのブロック図

【図133】 カラーチャンネルプロセッサにおける多機能ブロックのブロック図

【図134】 合成動作のためのブロック図

【図135】 スキャンラインの逆変換を示す図

【図136】 指定されたピクセルにおける値を計算するために必要なステップのブロック図

【図137】 画像変換エンジンのブロック図

【図138】 カーネルデスクリップションにおける2つのフォーマットを示す図

【図139】 b p フィールドの定義と解釈を示す図

【図140】 行列乗算を実行する乗算・加算部のブロック図

【図141】 コプロセッサでのキャッシュ及びキャッ

シュ制御部における制御、アドレス及びデータフローを示す図

【図142】 キャッシュのメモリ構成を示す図

【図143】 コプロセッサにおけるキャッシュ制御部のためのアドレスフォーマットを示す図

【図144】 カラーチャンネルプロセッサにおける多機能ブロックのブロック図

【図145】 図144のキャッシュ及びキャッシュコントローラのコプロセッサ入力インターフェーススイッチを示す図

【図146】 主アドレス及びデータパスを示すコプロセッサの4ポートダイナミックローカルメモリ制御部を示す図

【図147】 図146の制御部における状態機構図

【図148】 図146の仲裁部における機能を詳細にリストした擬似コードを示す図

【図149】 図146において用いられる要求者プライオリティビットの構造および用語を示す図

【図150】 コプロセッサにおける外部インターフェース制御部をより詳細に示す図

【図151】 コプロセッサで用いられる物理アドレスへのマッピング処理又は物理アドレスからのマッピング処理を示す図

【図152】 コプロセッサで用いられる物理アドレスへのマッピング処理又は物理アドレスからのマッピング処理を示す図

【図153】 コプロセッサで用いられる物理アドレスへのマッピング処理又は物理アドレスからのマッピング処理を示す図

【図154】 コプロセッサで用いられる物理アドレスへのマッピング処理又は物理アドレスからのマッピング処理を示す図

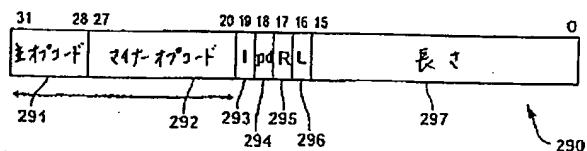
【図155】 図150におけるI B u s受信部をより詳細に示す図

【図156】 図2におけるR B u s受信部をより詳細に示す図

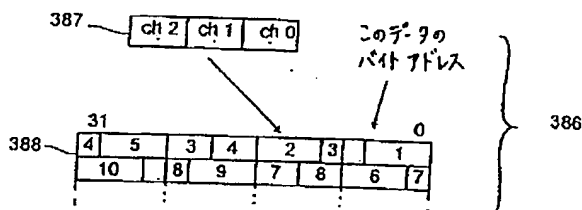
【図157】 図150におけるメモリ管理部をより詳細に示す図

【図158】 図2における周辺インターフェース制御部をより詳細に示す図

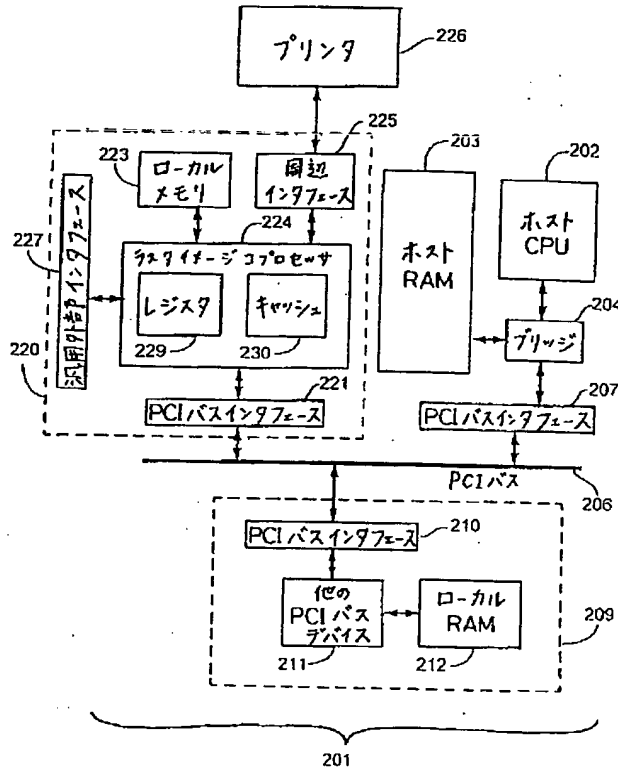
【図11】



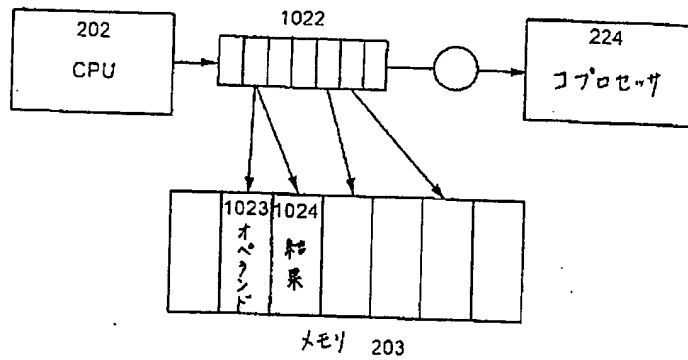
【図23】



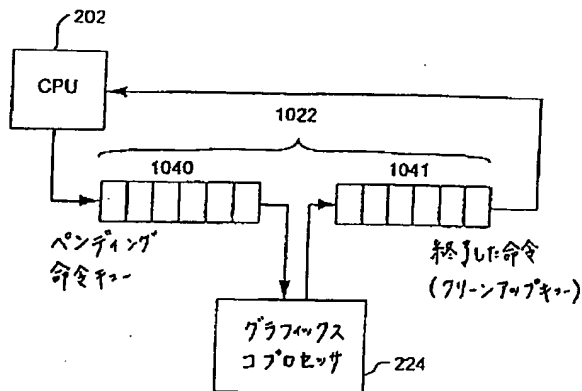
【図1】



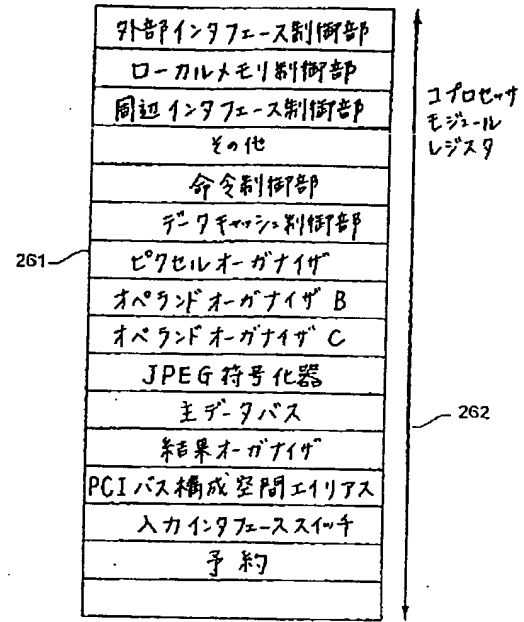
【図4】



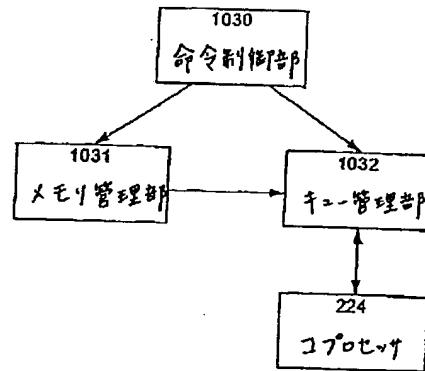
【図6】



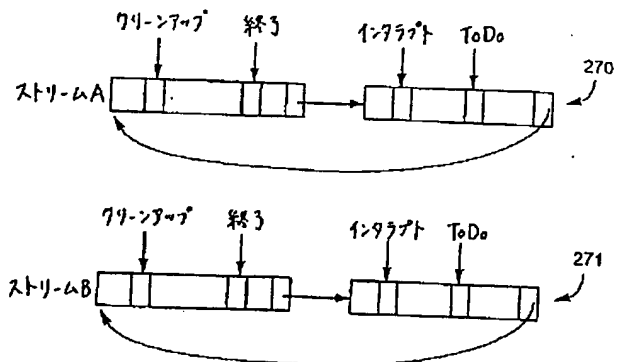
【図3】



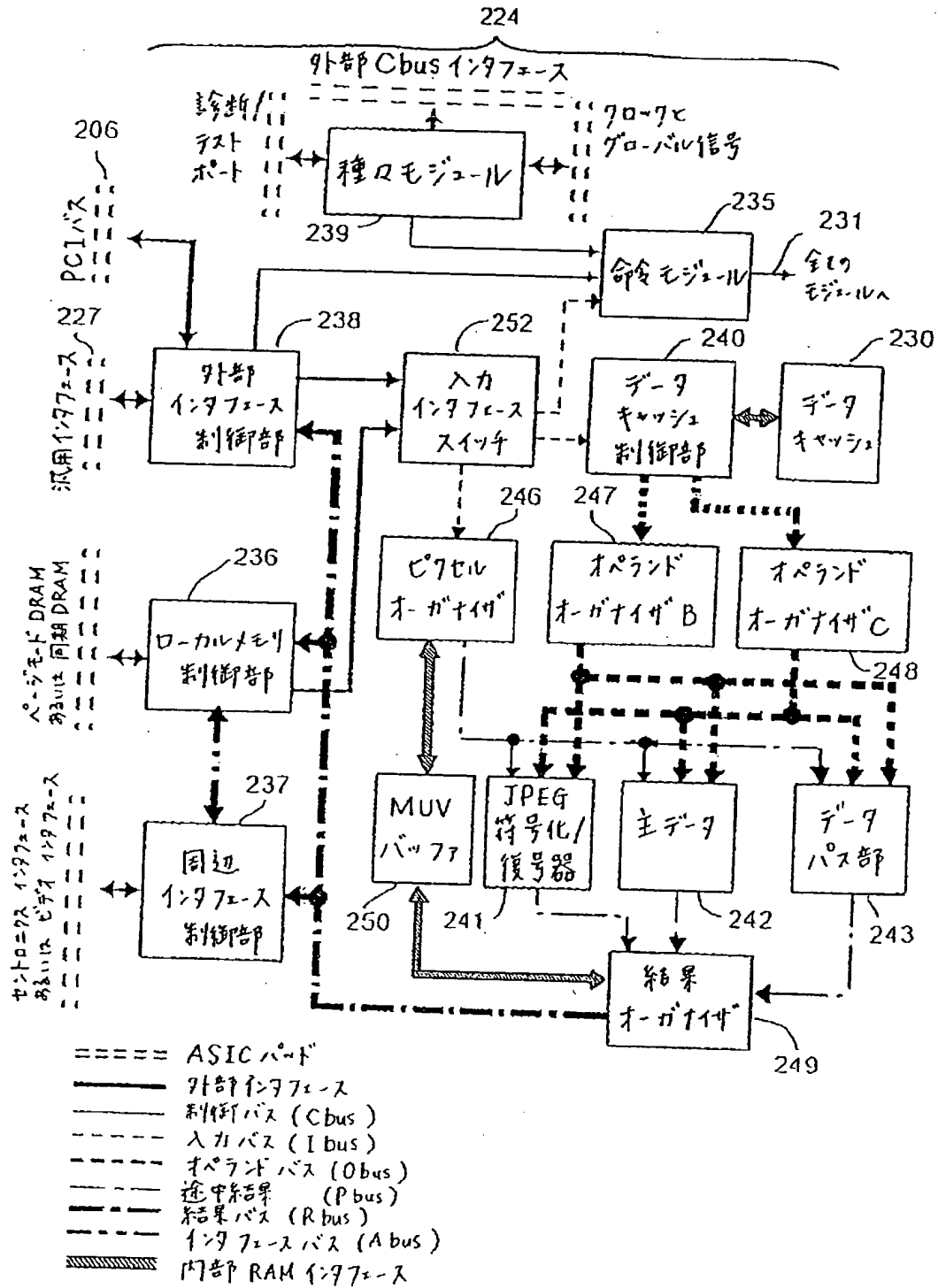
【図5】



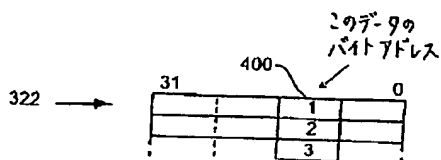
【図8】



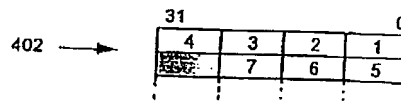
【図2】



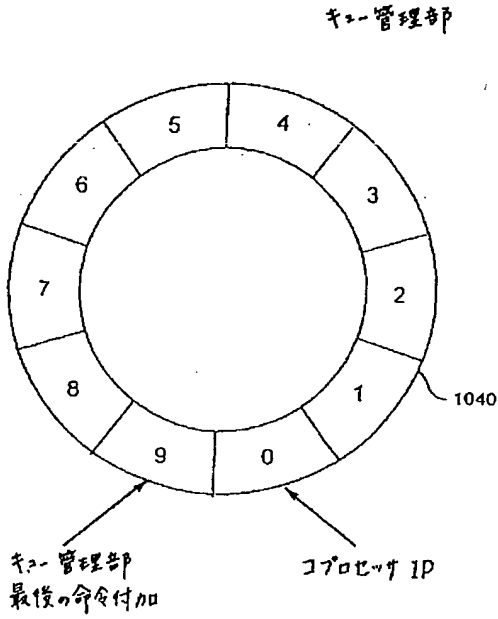
【図26】



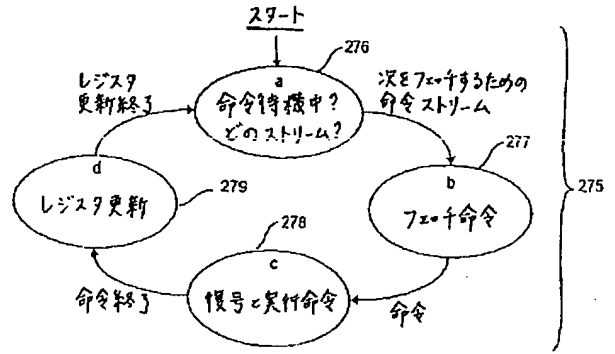
【図27】



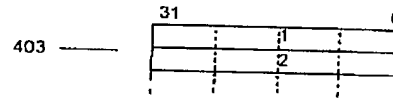
【図7】



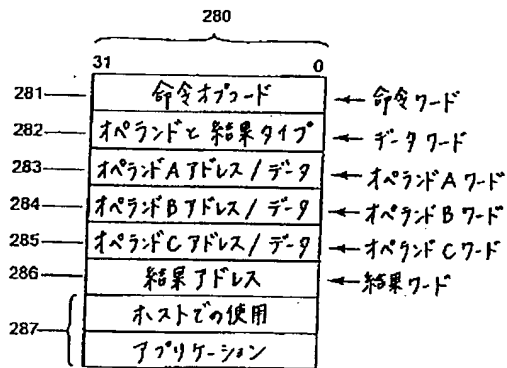
【図9】



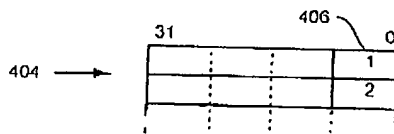
【図28】



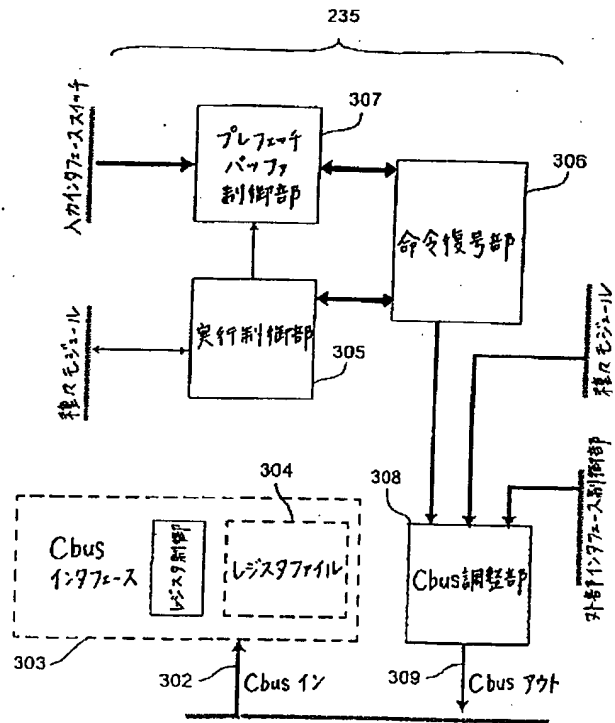
【図10】



【図29】



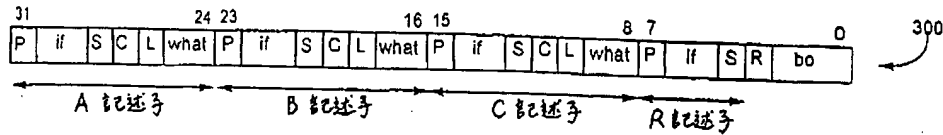
【図13】



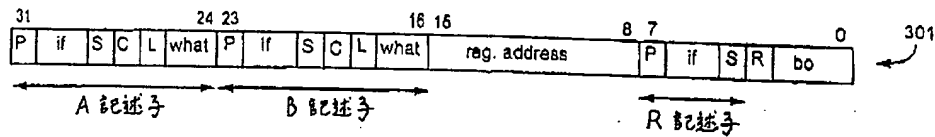
【図12】

データワードのフォーマット

3オペランド命令:

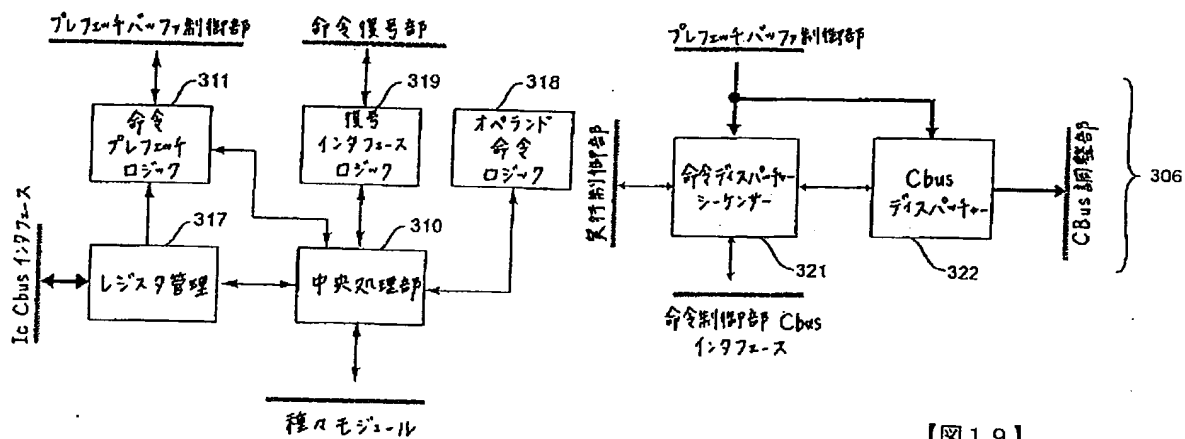


2(以下)オペランド命令:



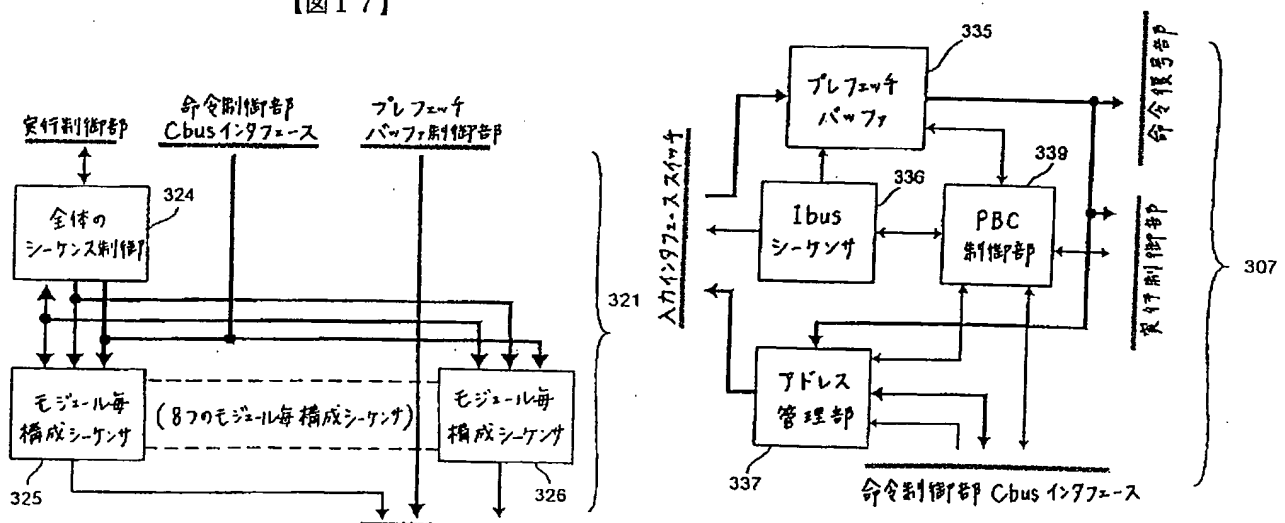
【図14】

【図16】

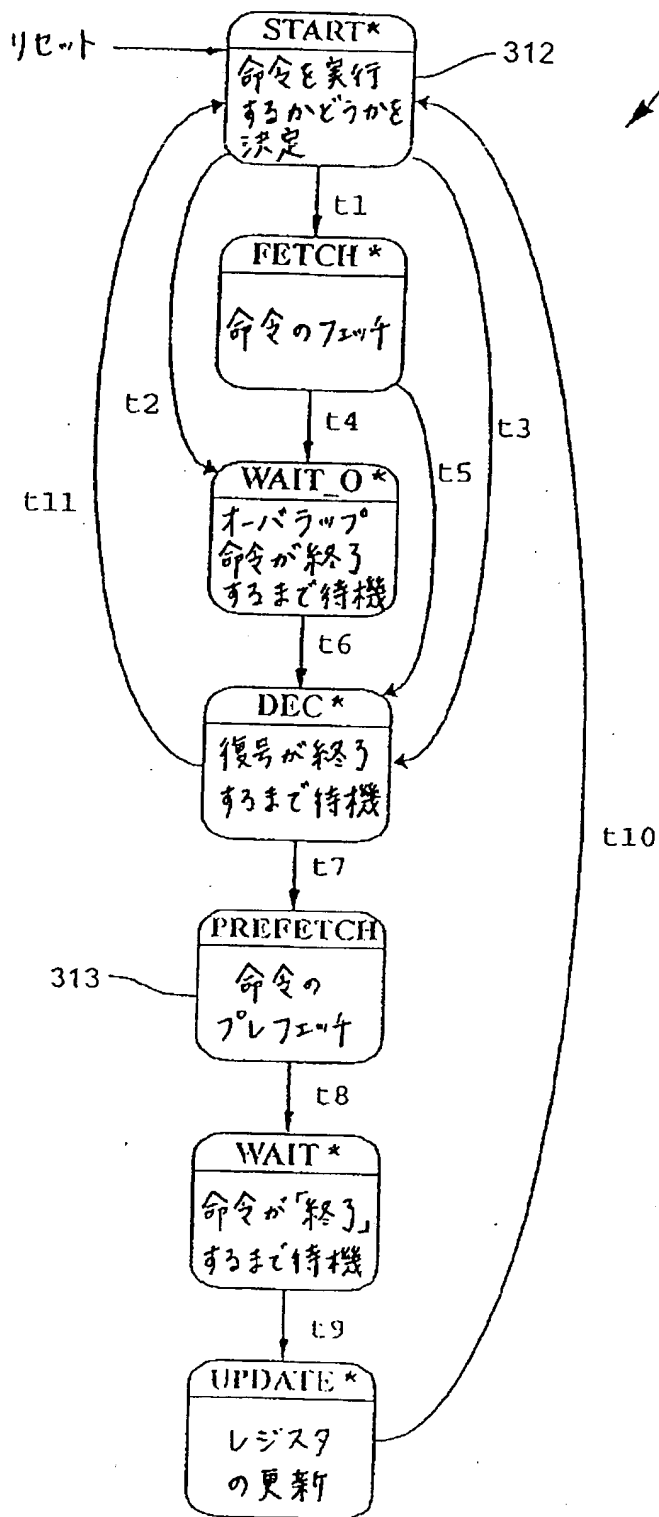


【図17】

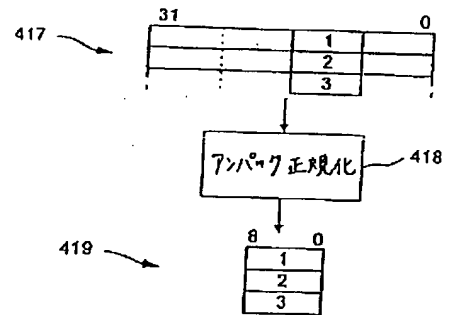
【図19】



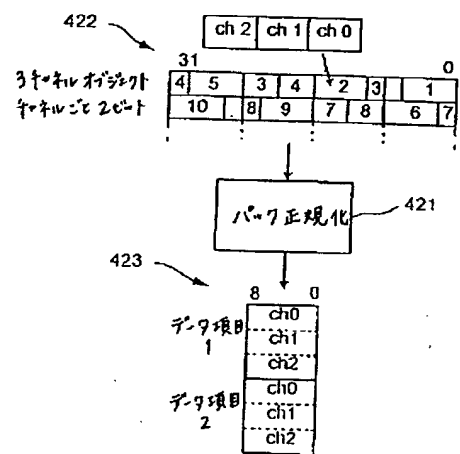
【図15】



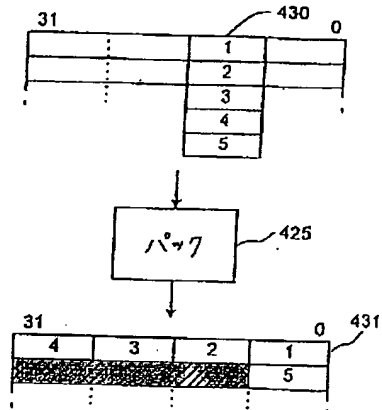
【図33】



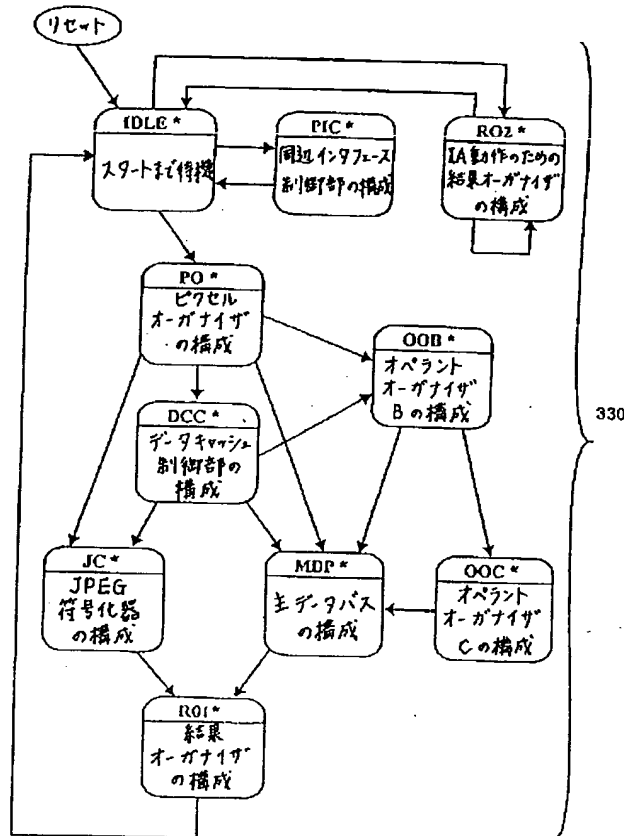
【図34】



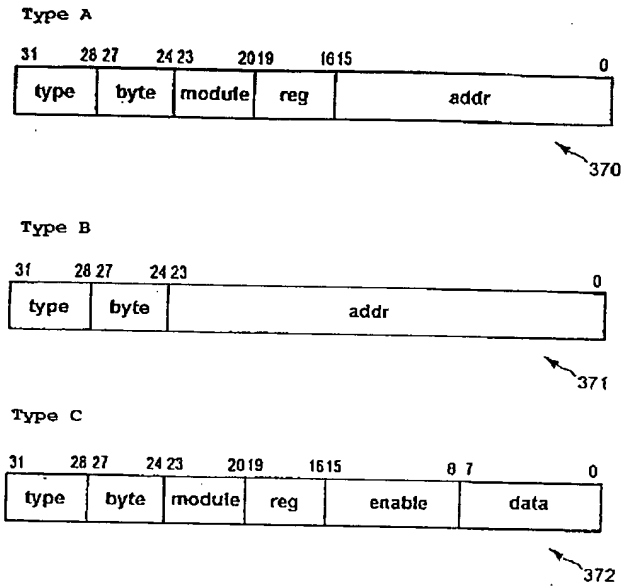
【図35】



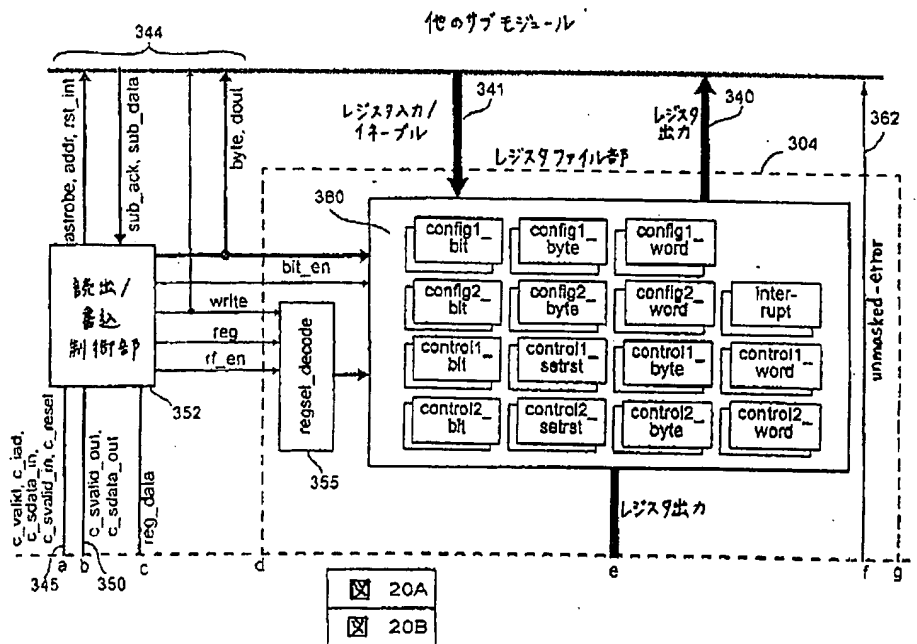
【図 18】



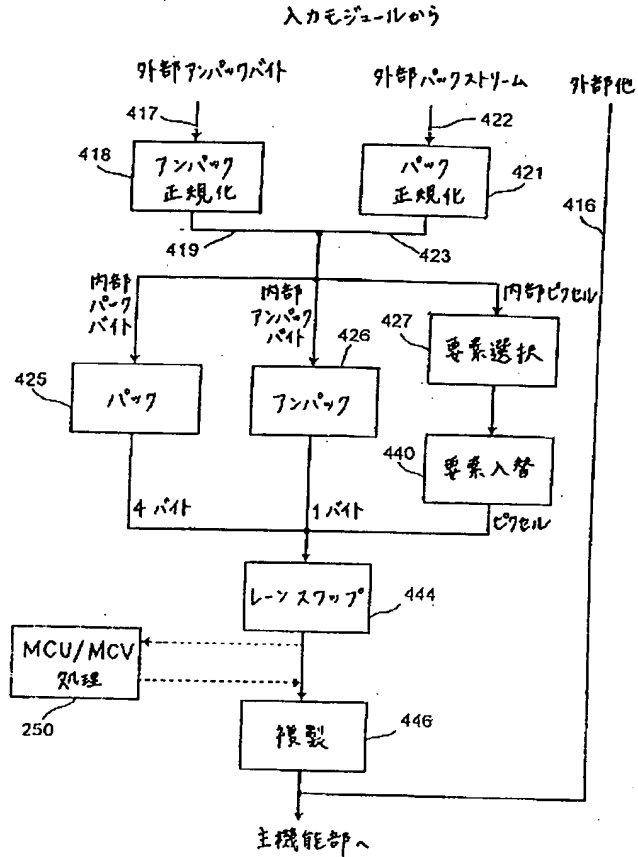
【図 21】



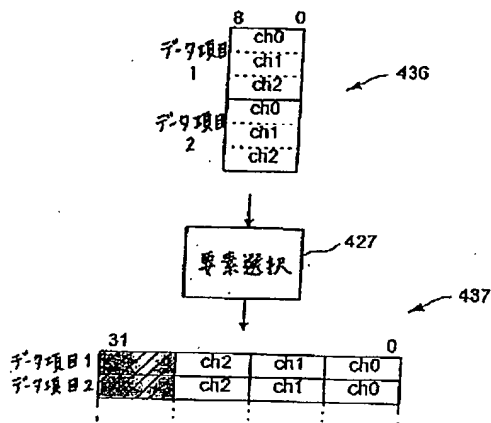
【図 20 A】



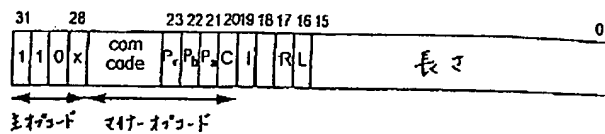
【図32】



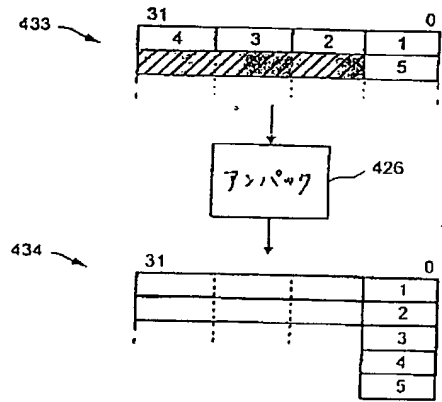
【図37】



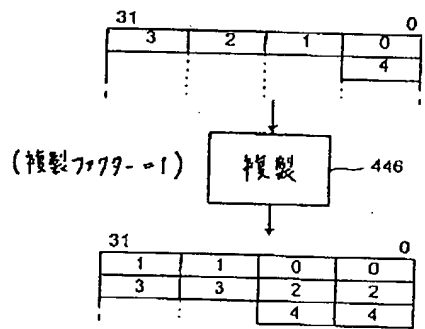
【図52】



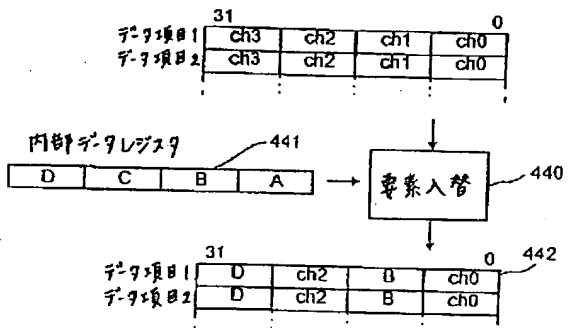
【図36】



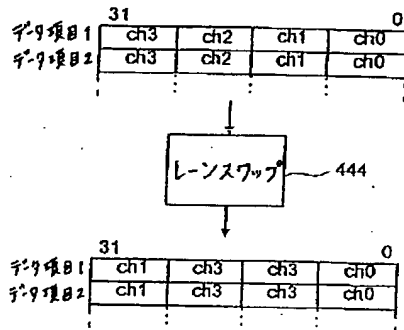
【図41】



【図38】

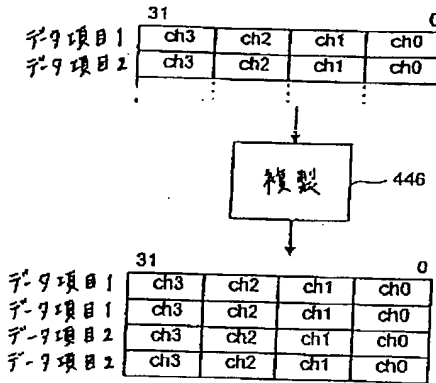


【図 3 9】

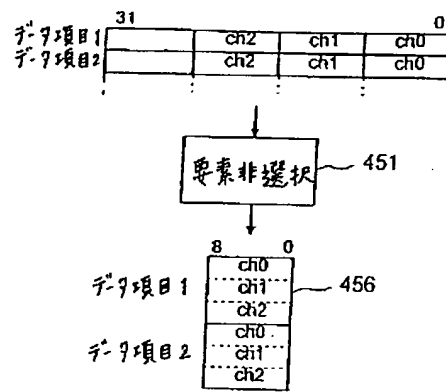
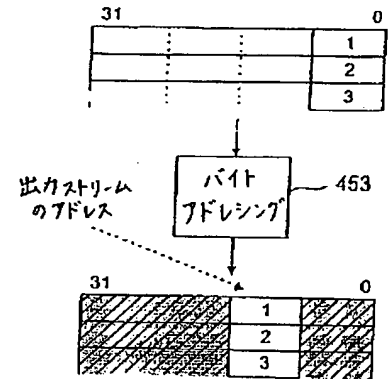


【図 4 2】

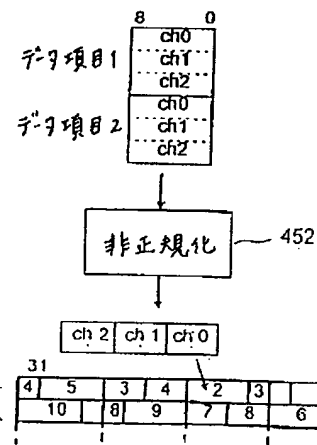
【図 4 0】



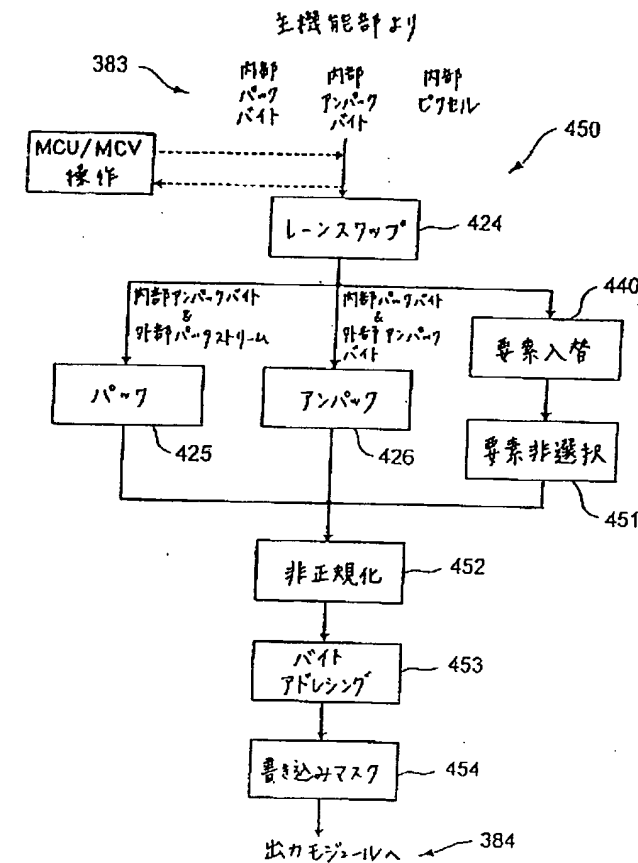
【図 4 3】



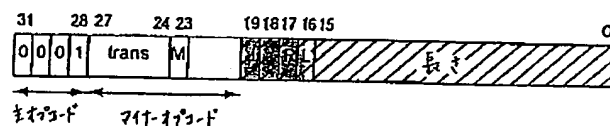
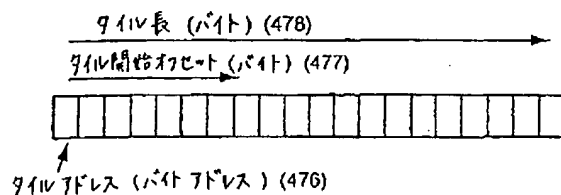
【図 4 4】



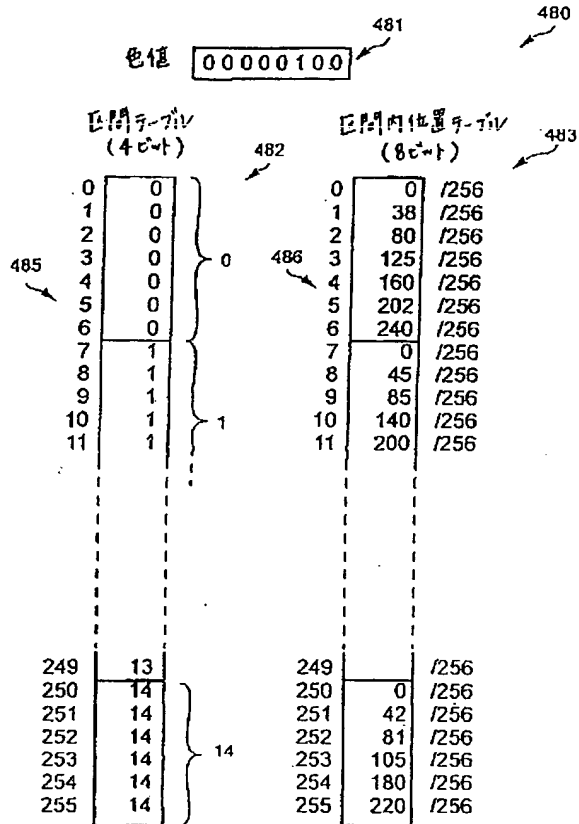
【図 6 4】



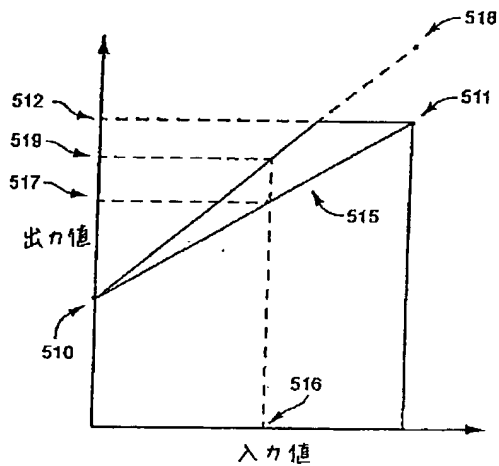
【図 5 5】



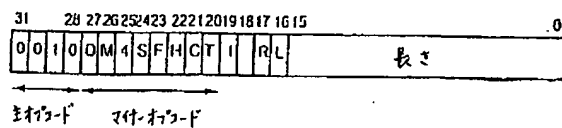
【図 5 6】



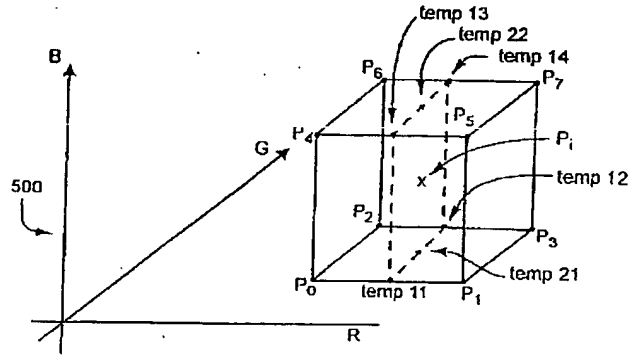
【図 5 9】



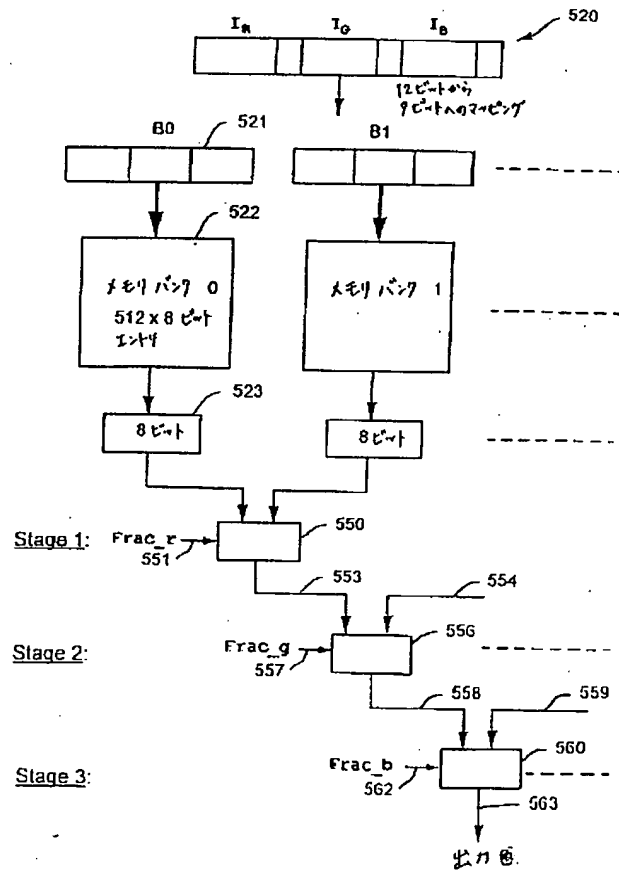
【図 7 6】



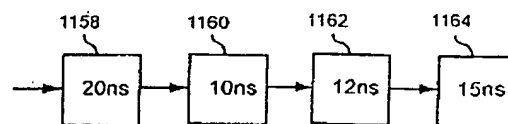
【図 5 8】



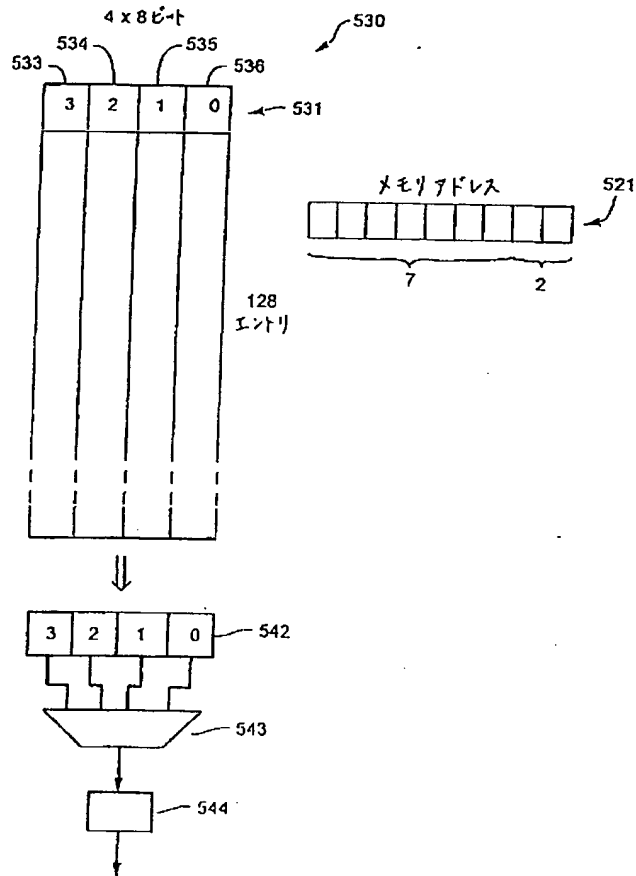
【図 6 0】



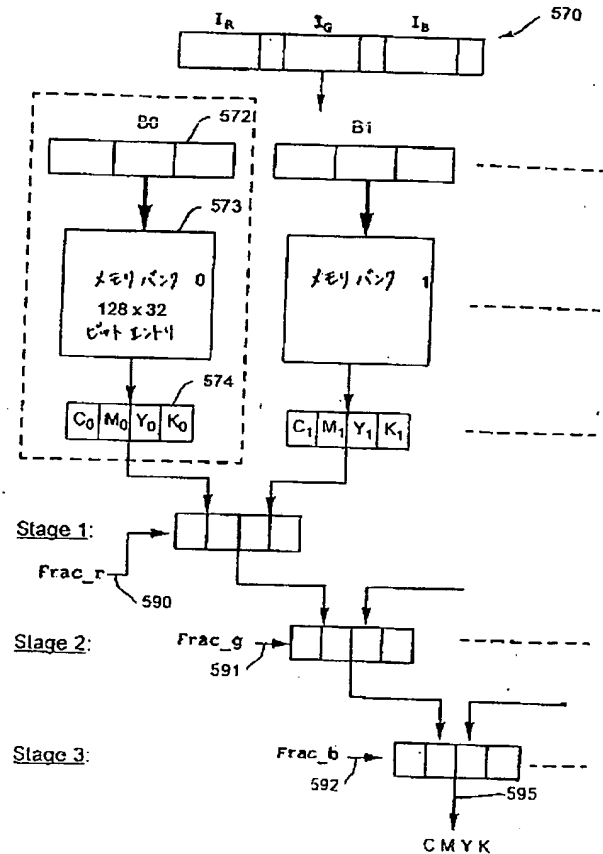
【図 8 1】



【図 6 1】

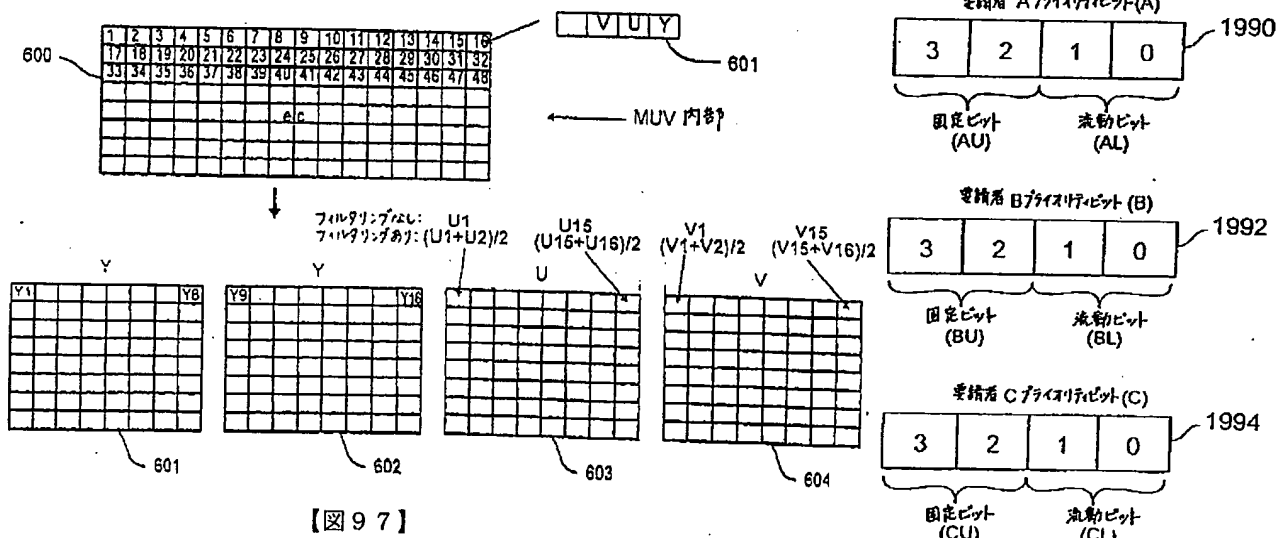


【図 6 2】

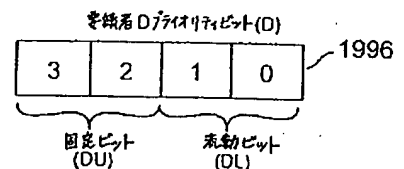
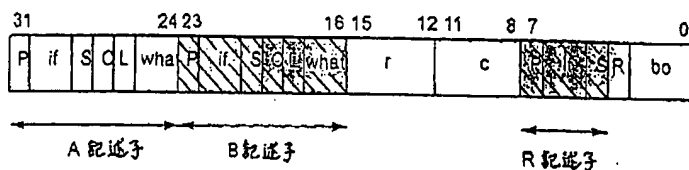


【図 6 6】

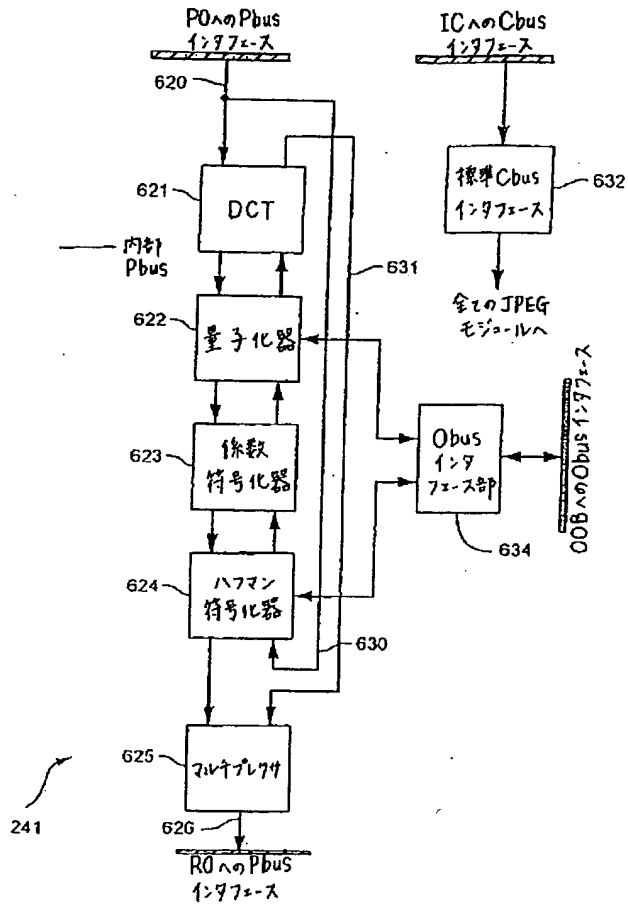
【図 1 4 9】



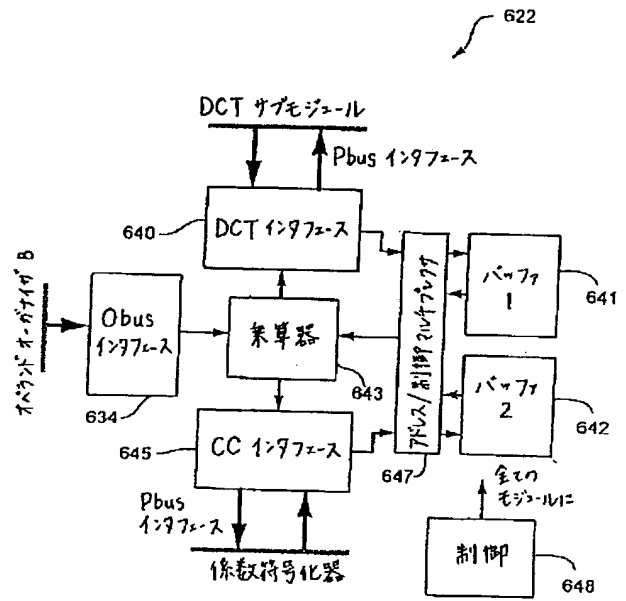
【図 9 7】



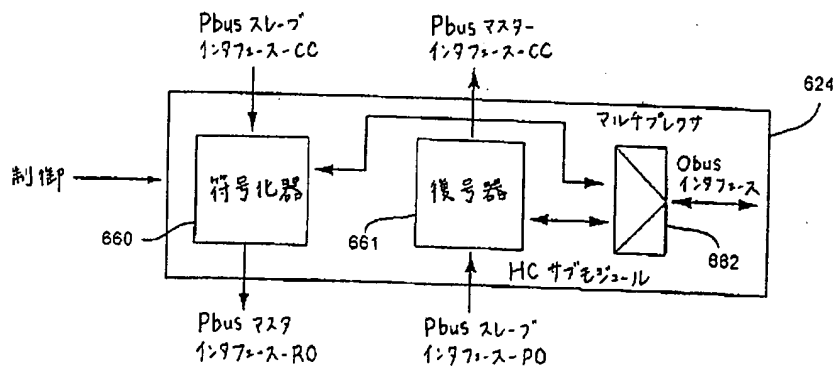
【図68】



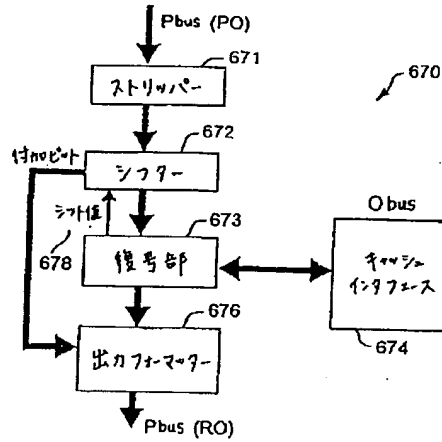
【図69】



【図70】

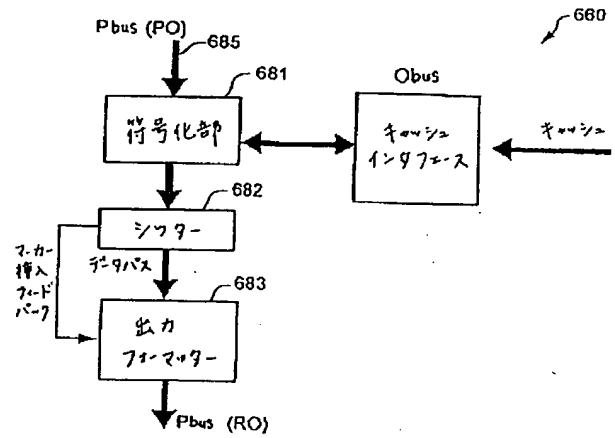


【図 7 1】

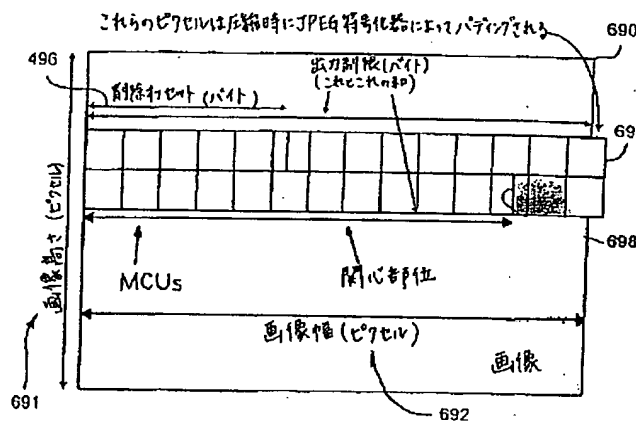


【図 7 3】

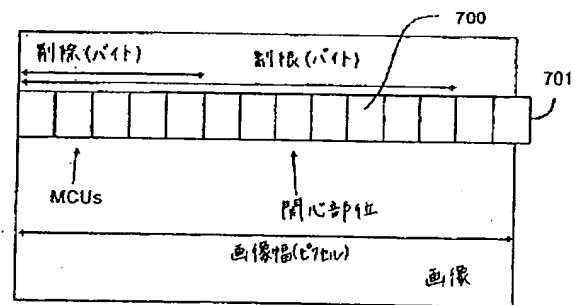
【図 7 2】



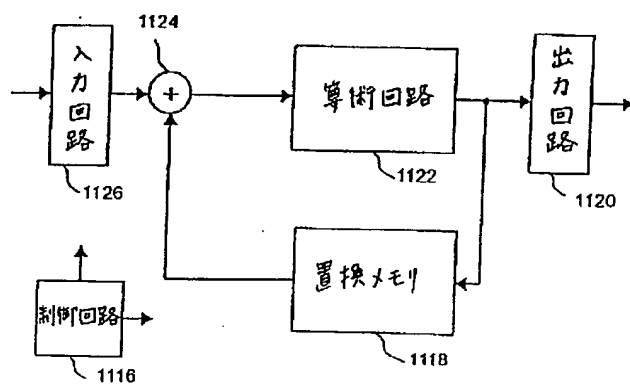
【図 7 4】



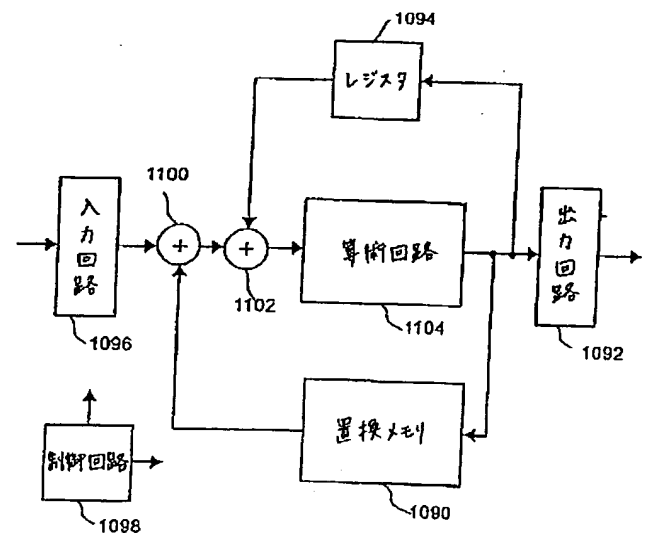
【図 7 9】



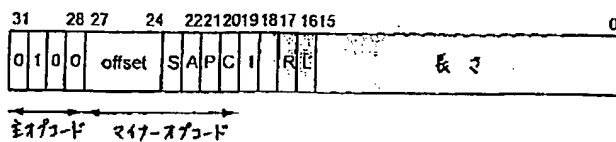
【図 7 7】



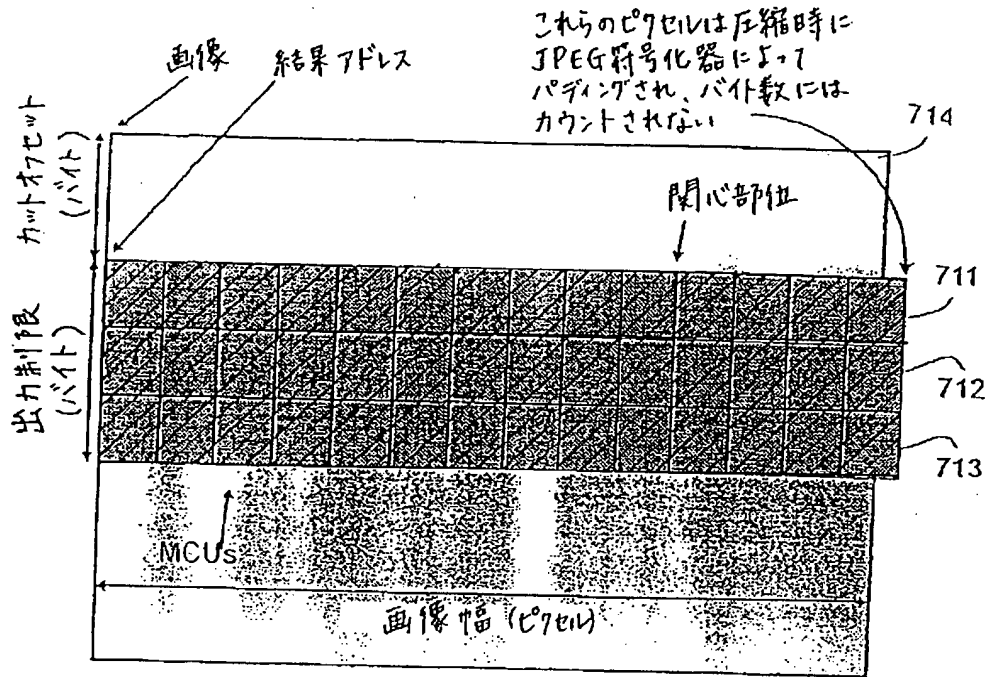
【図 1 0 0】



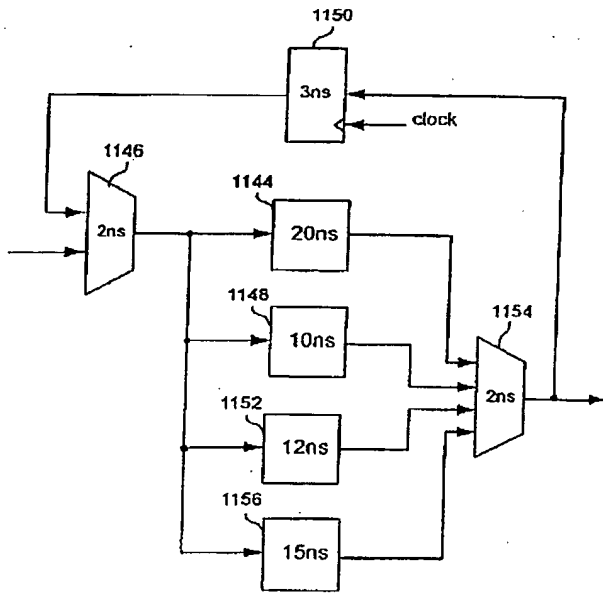
(従来例)



【図75】

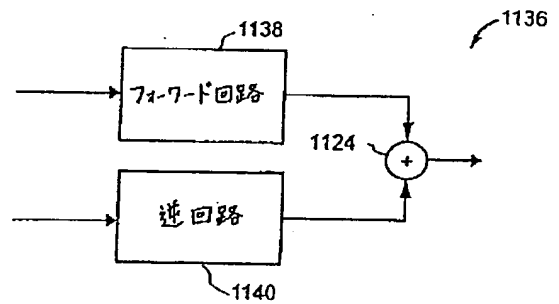


【図78】

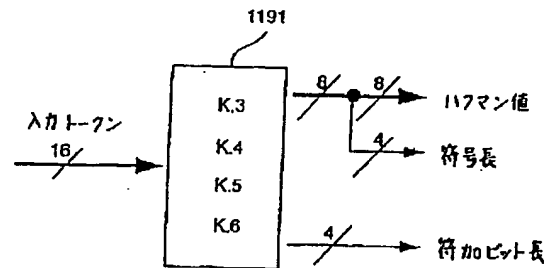


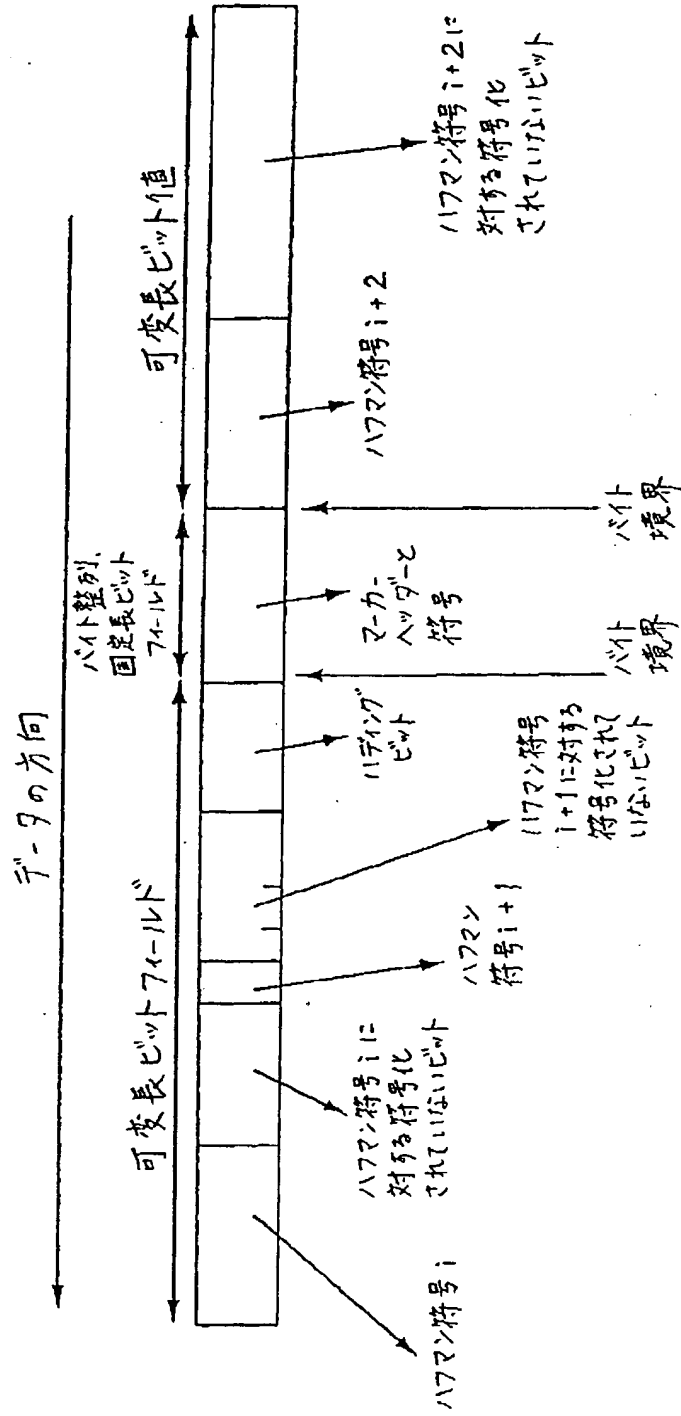
(従来例)

【図80】

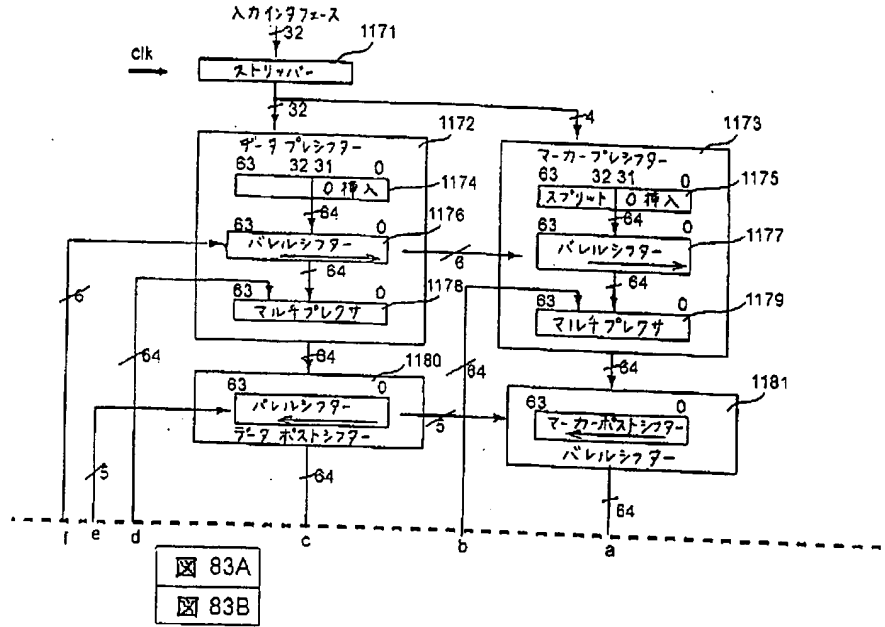


【図89】

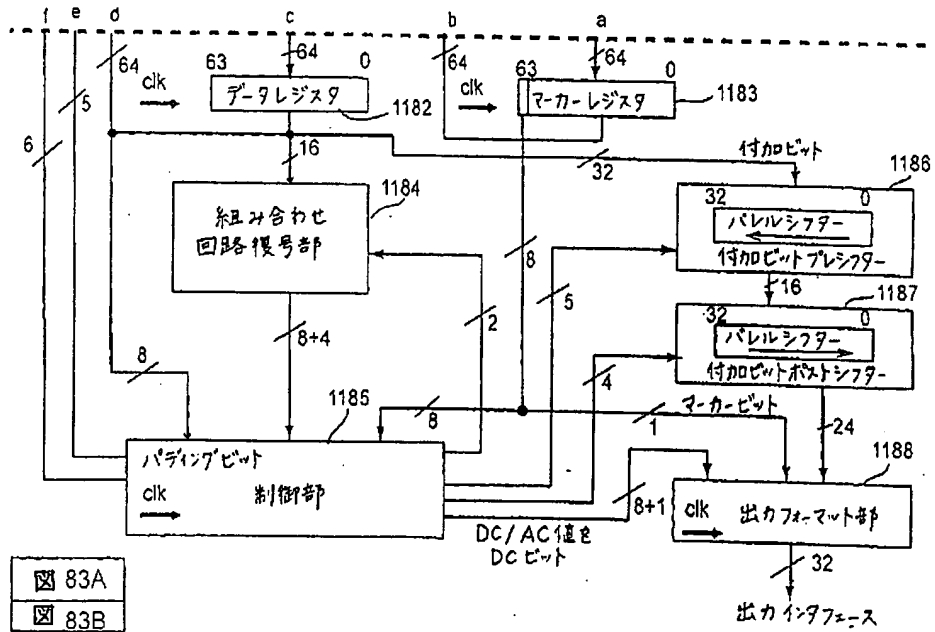




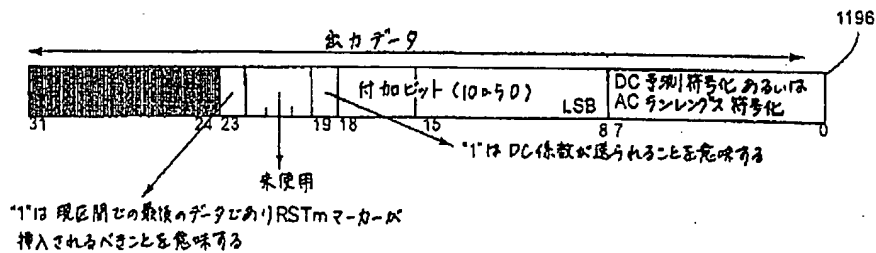
【図83A】



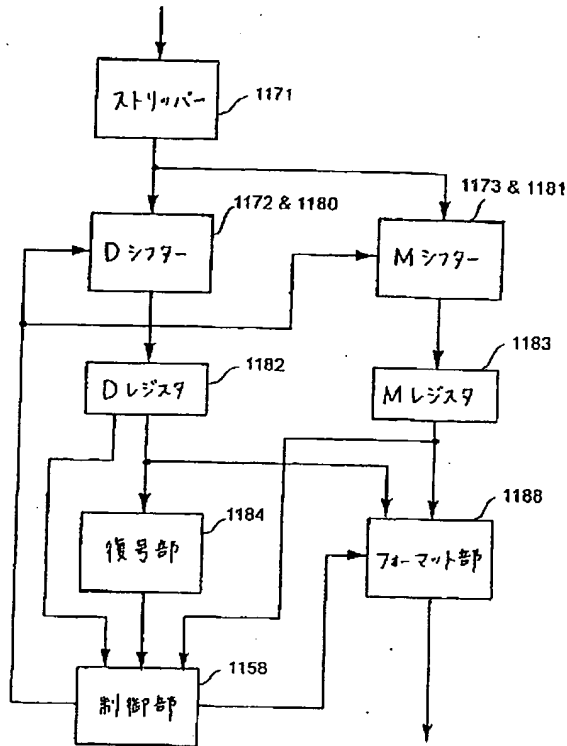
【図83B】



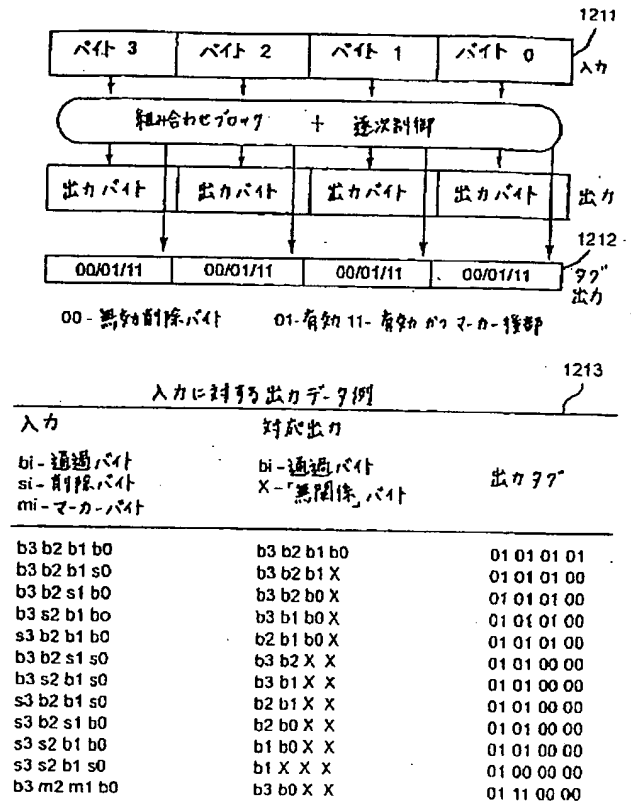
【図91】



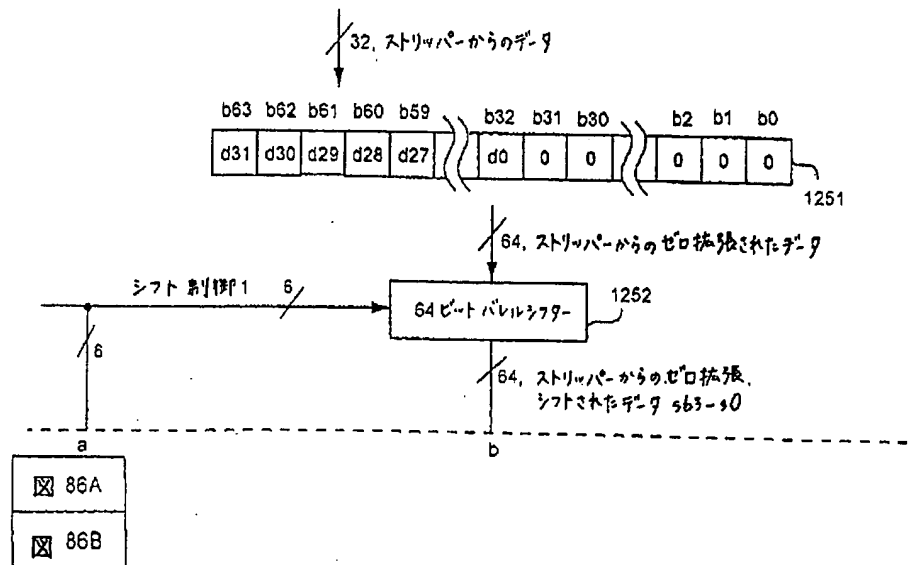
【図84】



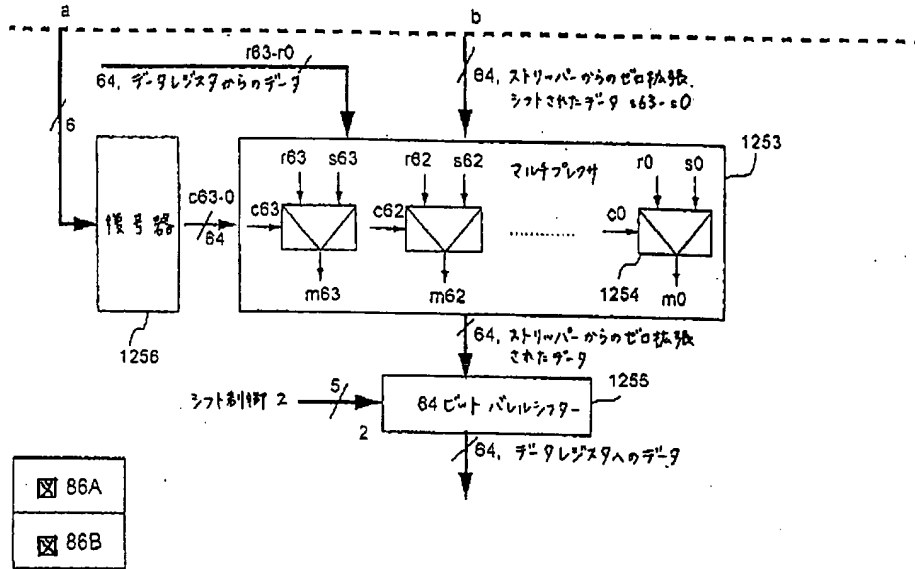
【図85】



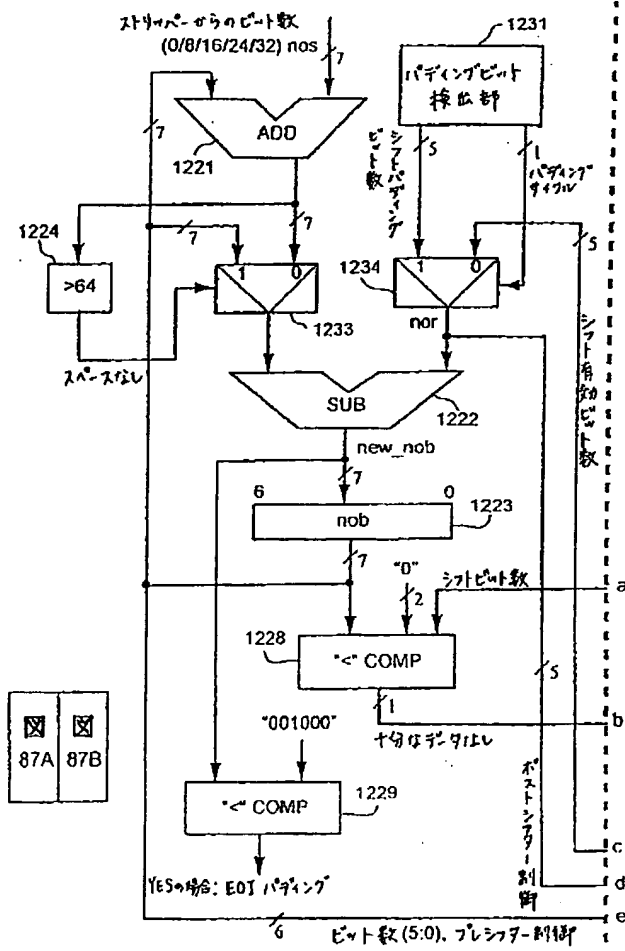
【図86A】



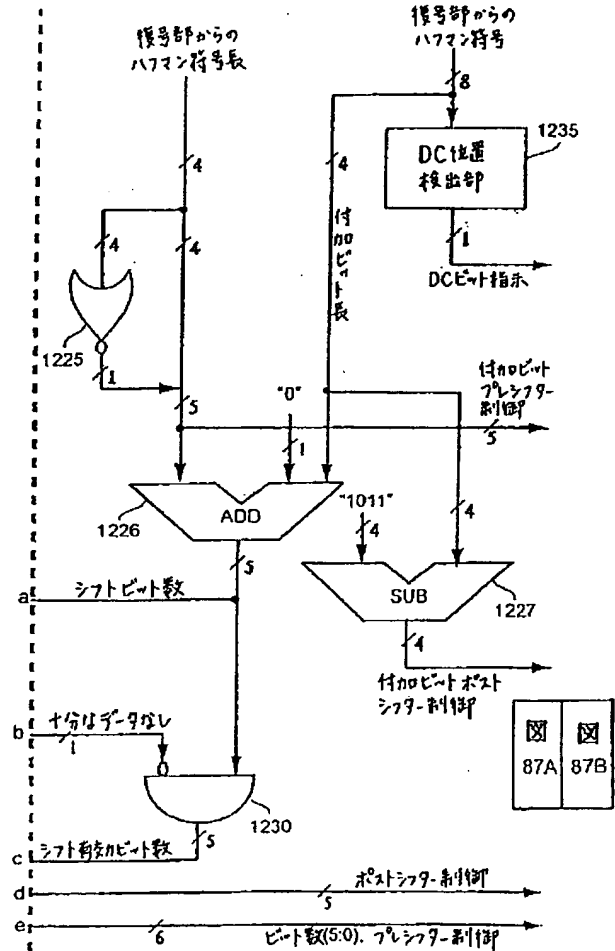
【図86B】



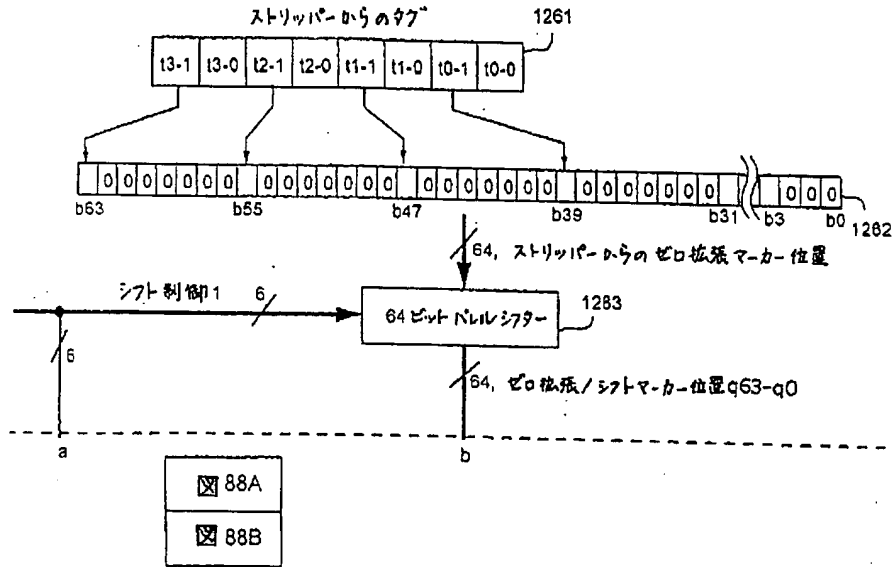
【図87A】



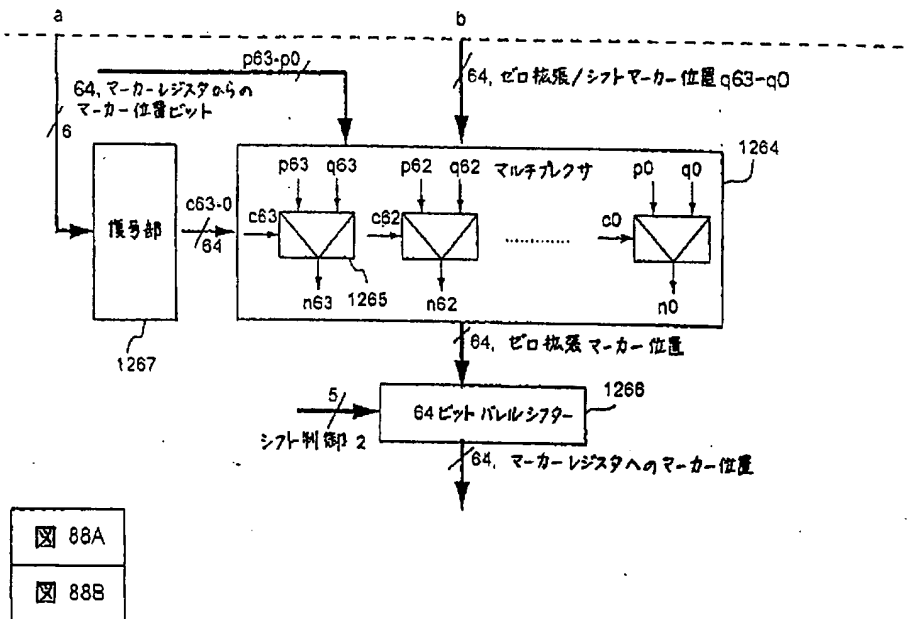
【図87B】



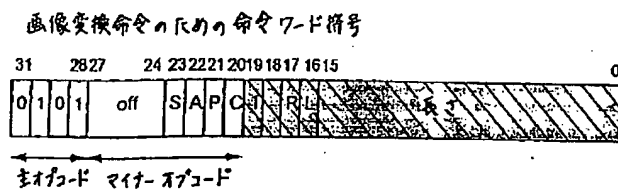
【図 8 8 A】



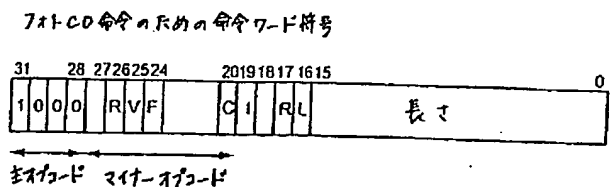
【図 8 8 B】



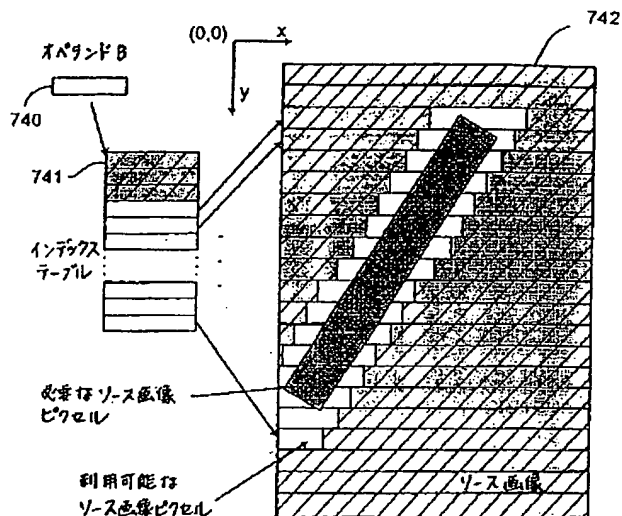
【図 9 3】



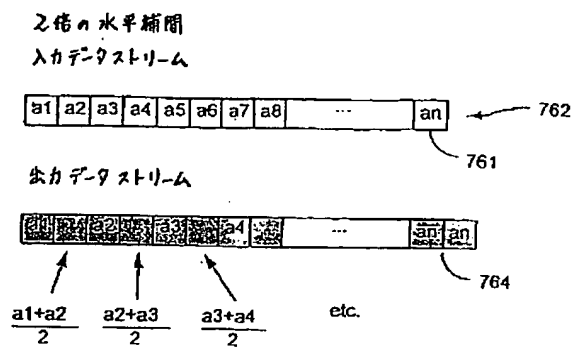
【図 1 0 6】



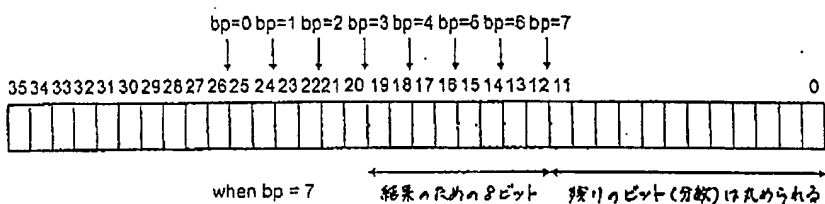
【図96】



【図102】

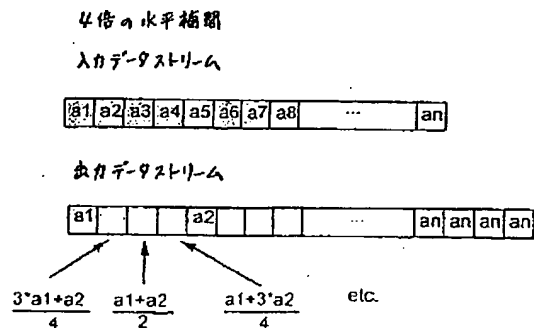


【図98】

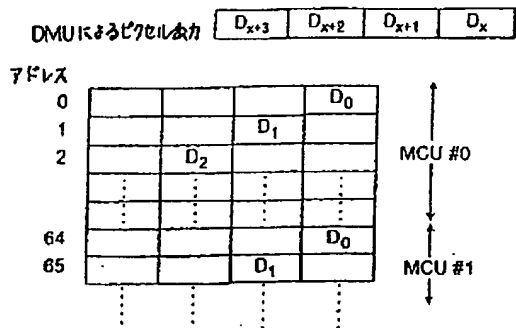
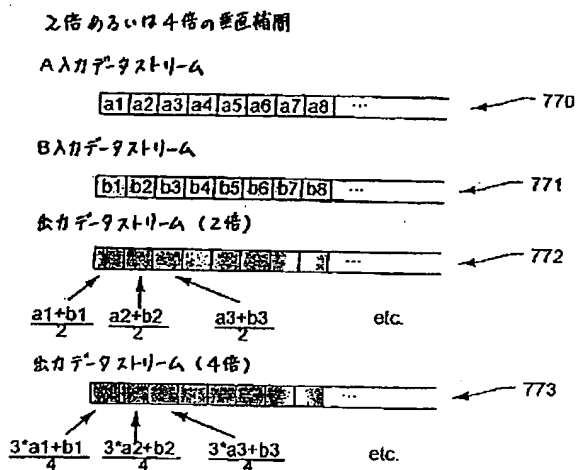


【図103】

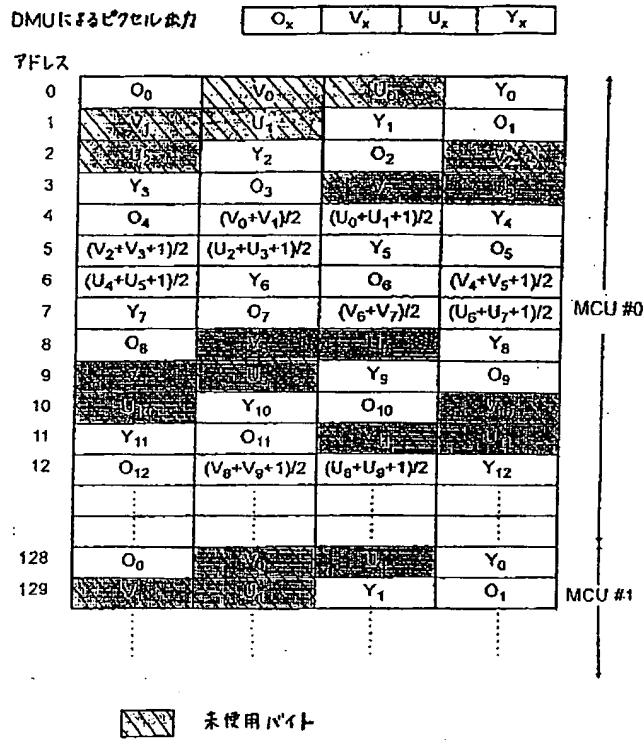
【図104】



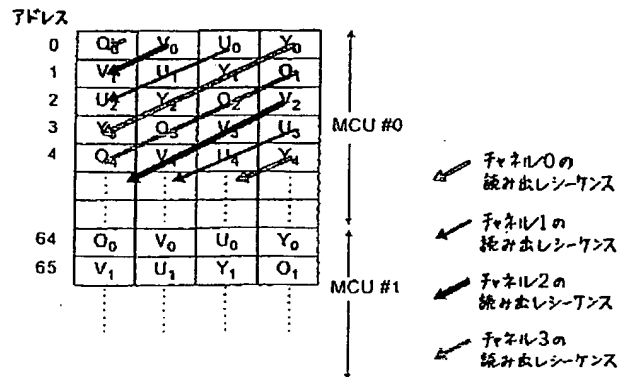
【図113】



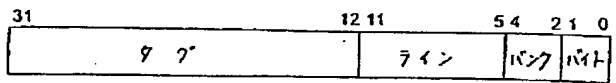
【図112】



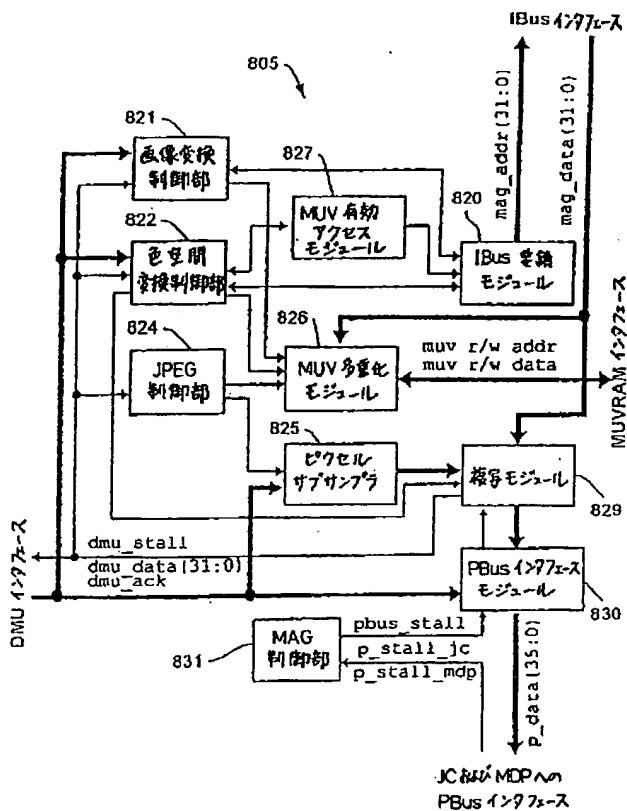
【図114】



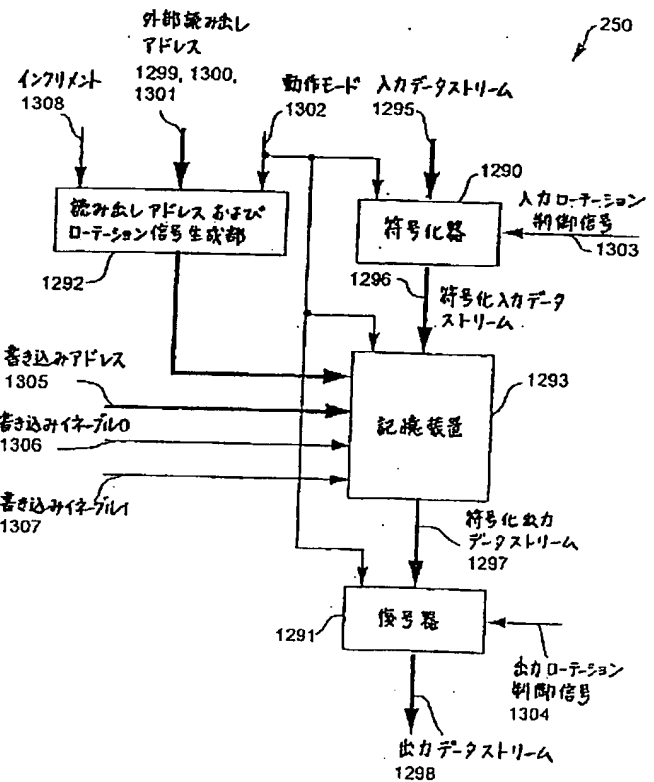
【図143】



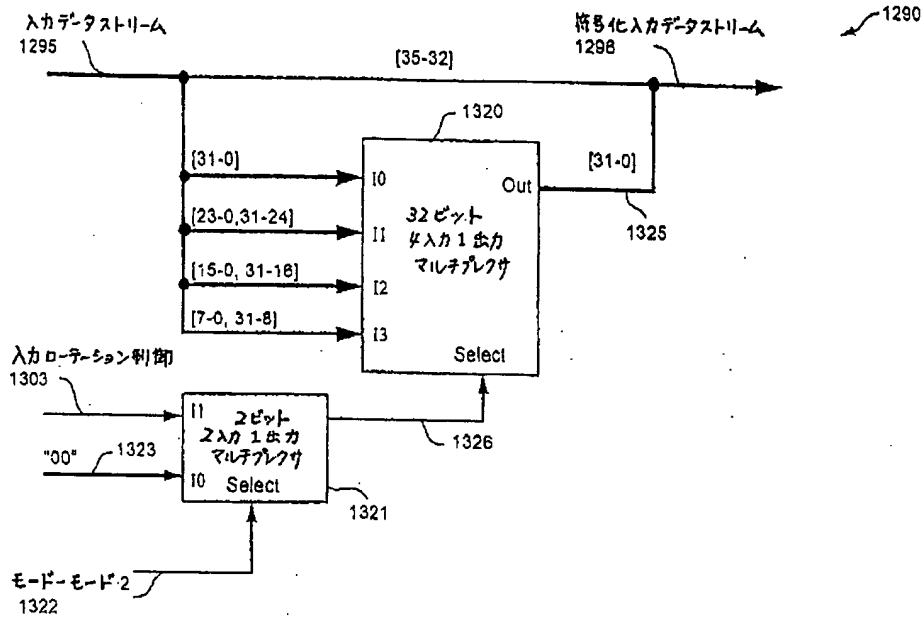
【図115】



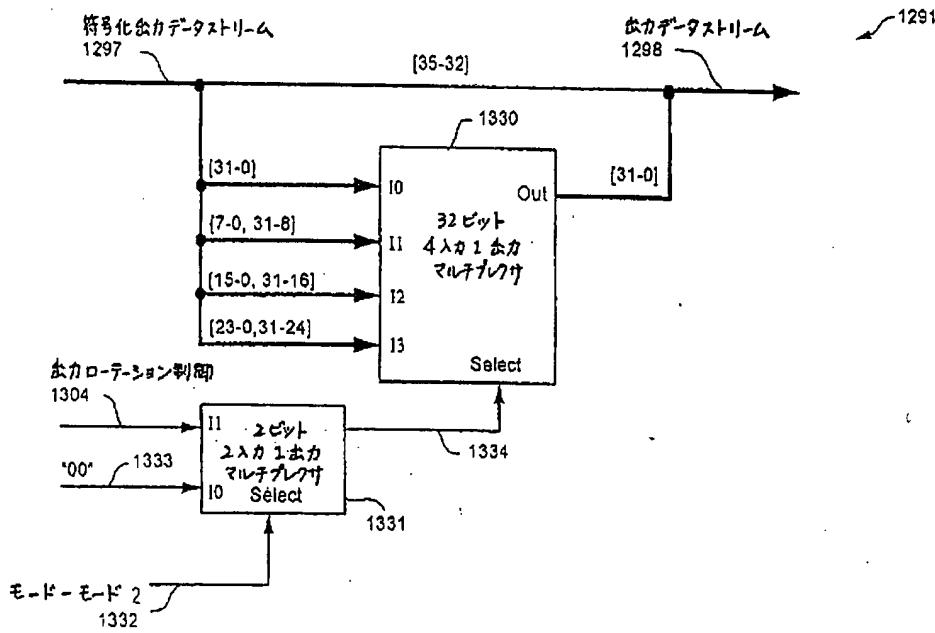
【図116】



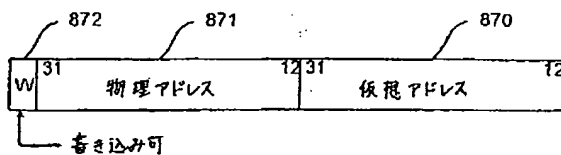
【図 1 1 7】



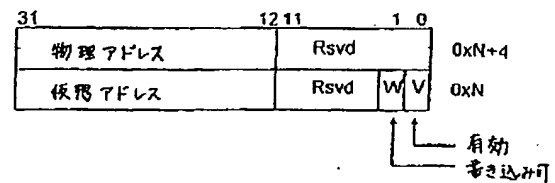
【図 1 1 8】



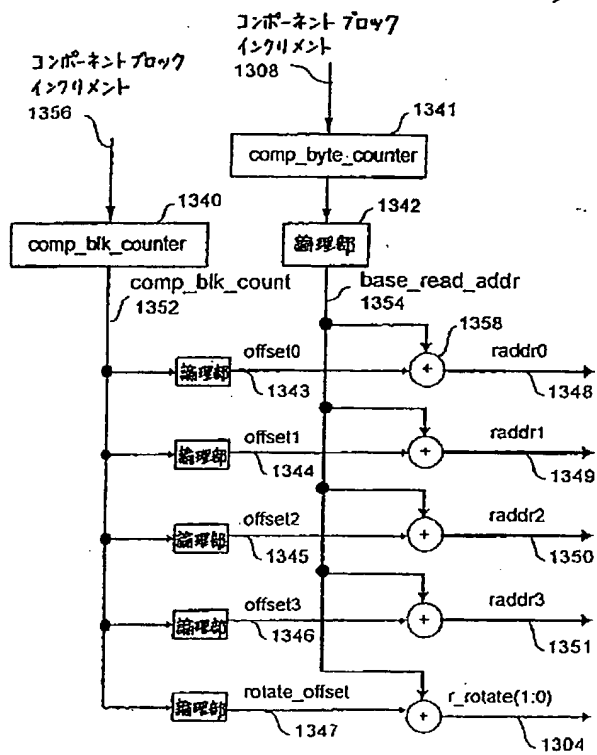
【図 1 5 1】



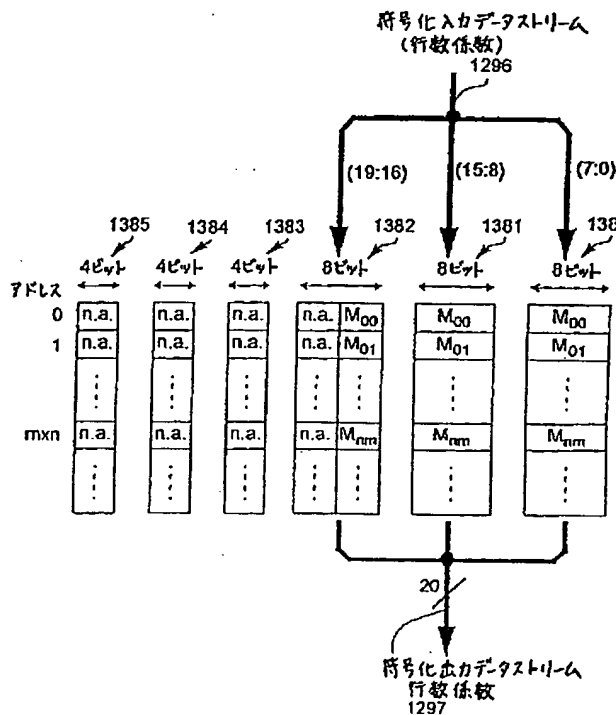
【図 1 5 3】



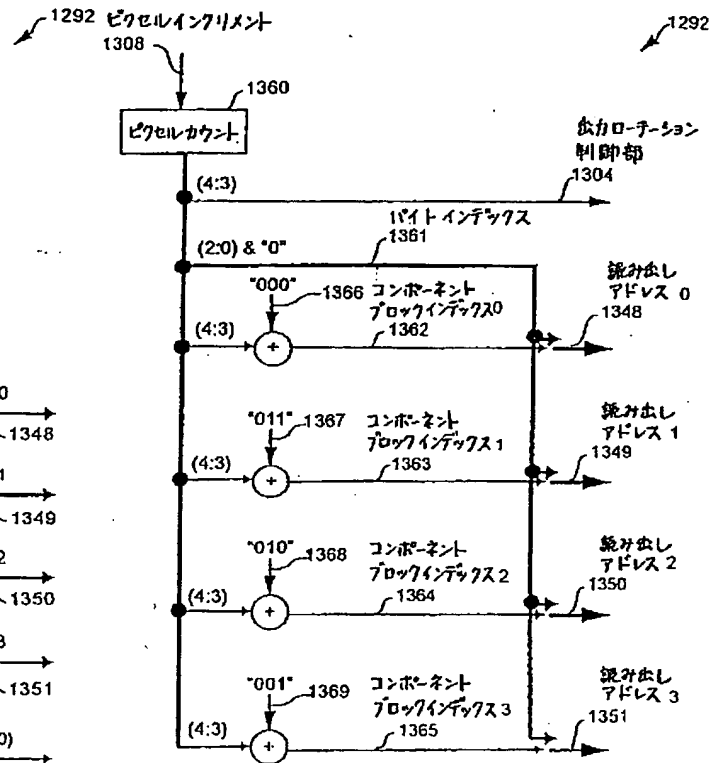
【図119】



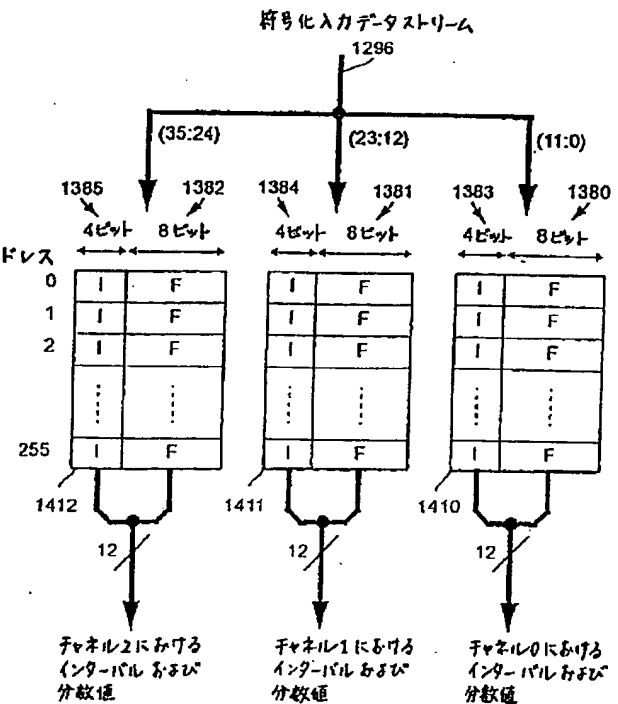
【図123】



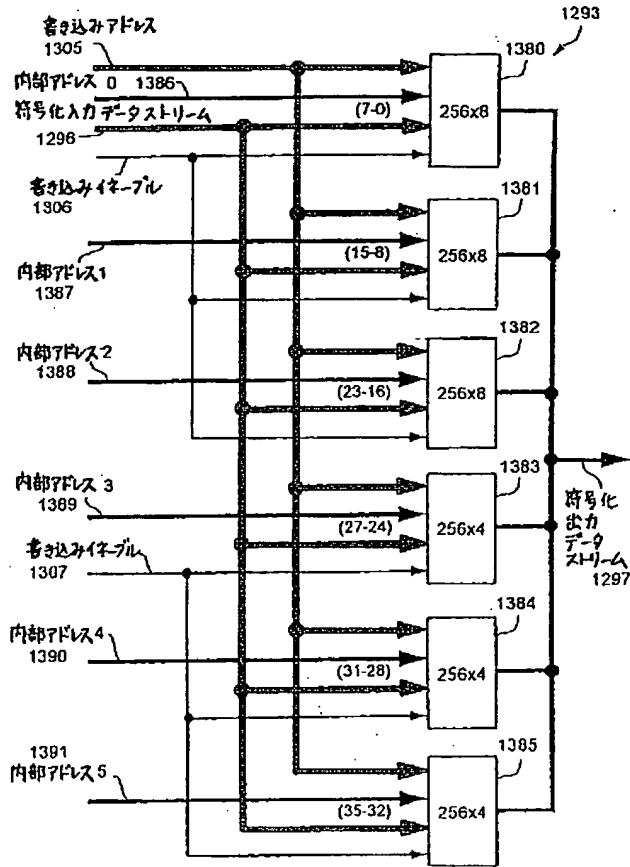
【図120】



【図124】

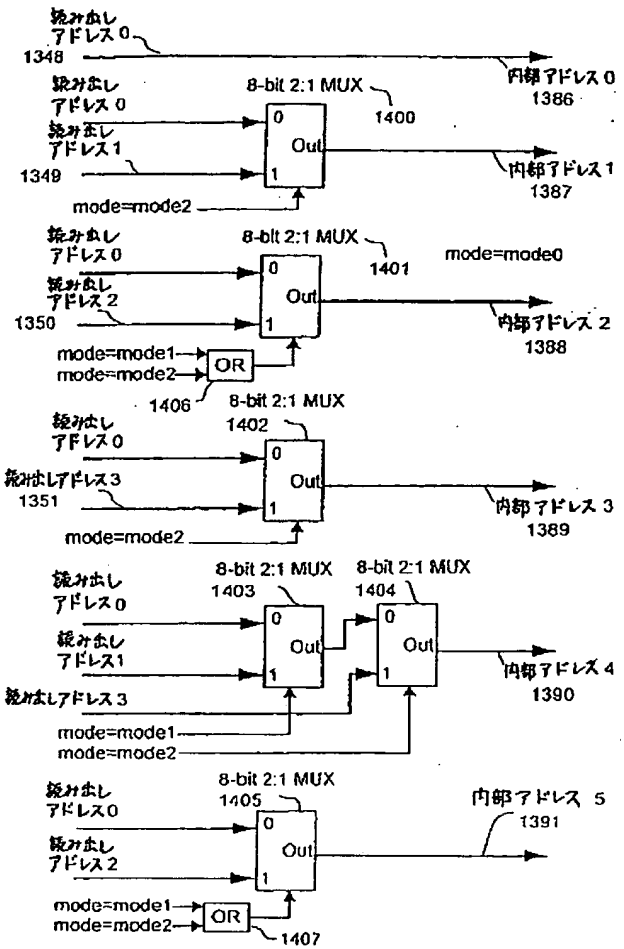


【図121】

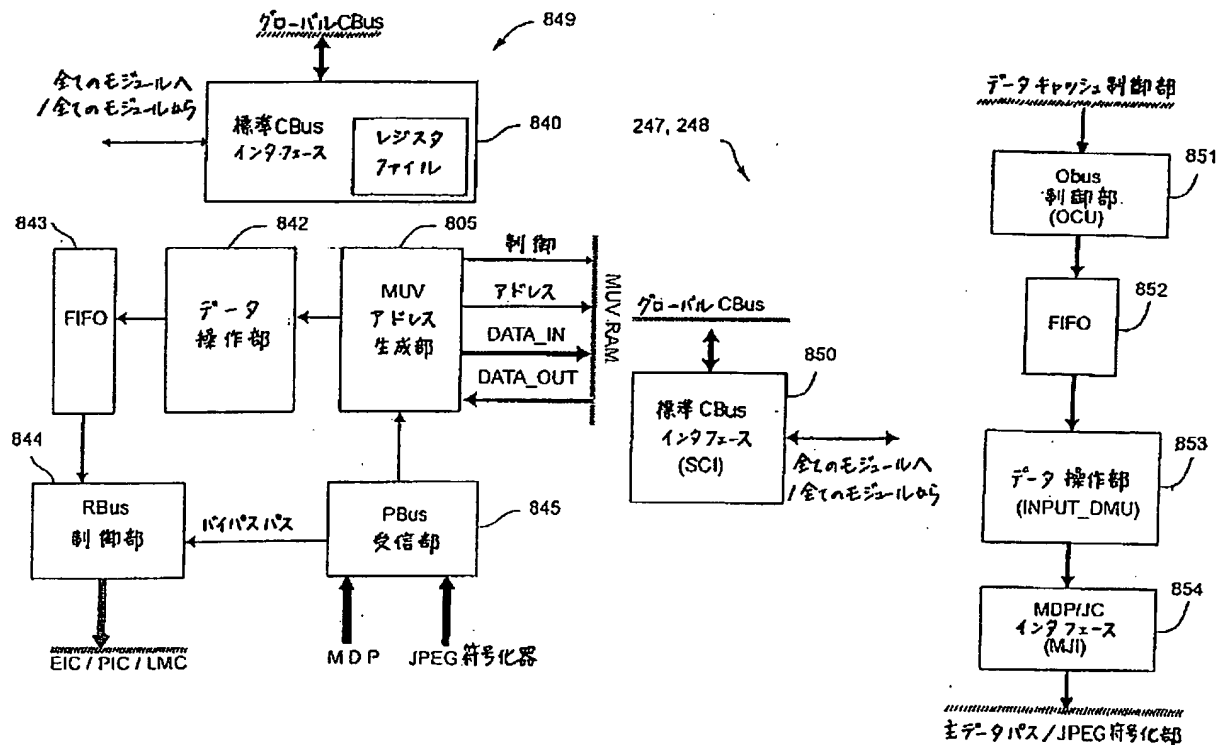


【図127】

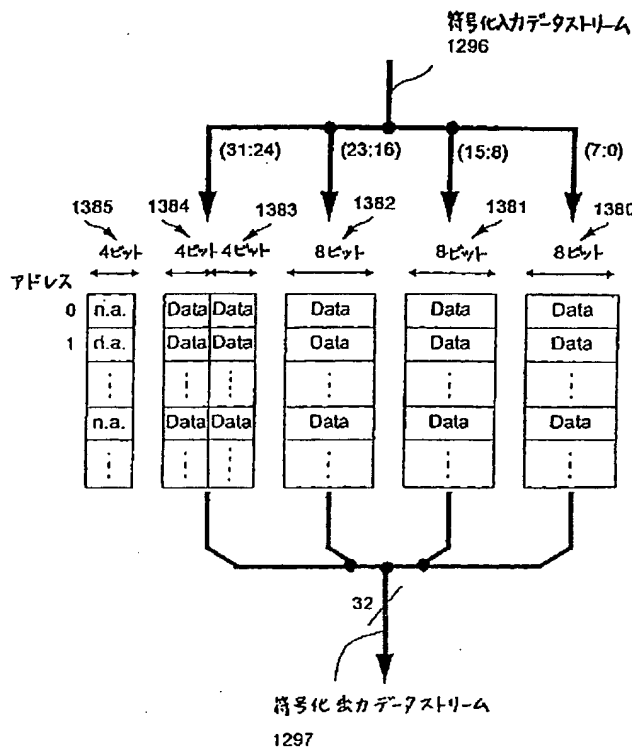
【図122】



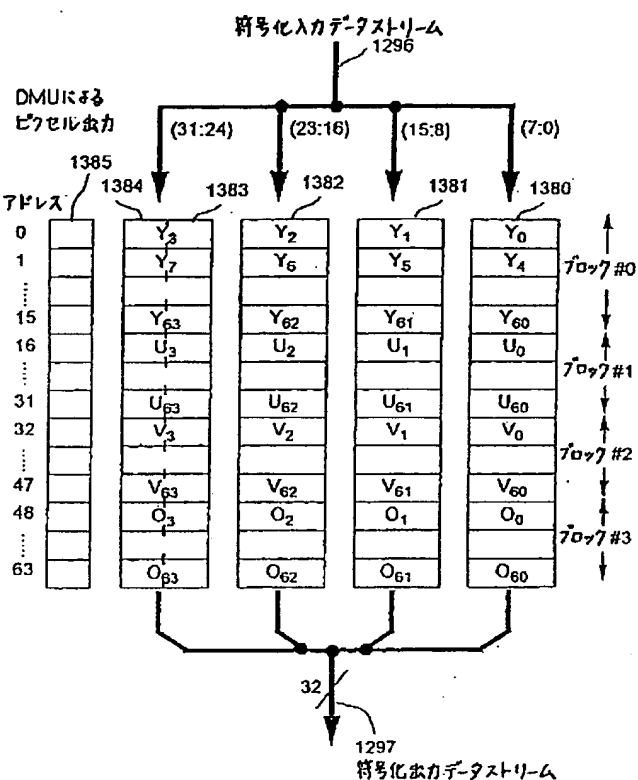
【図128】



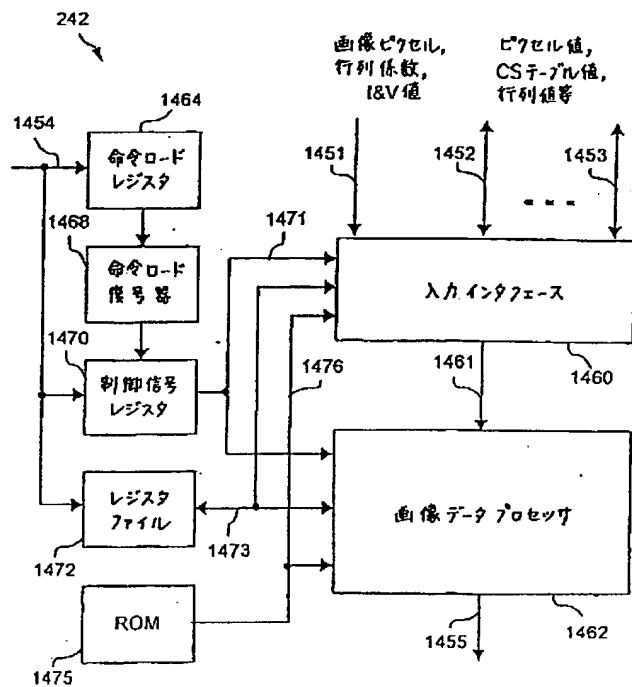
【図 125】



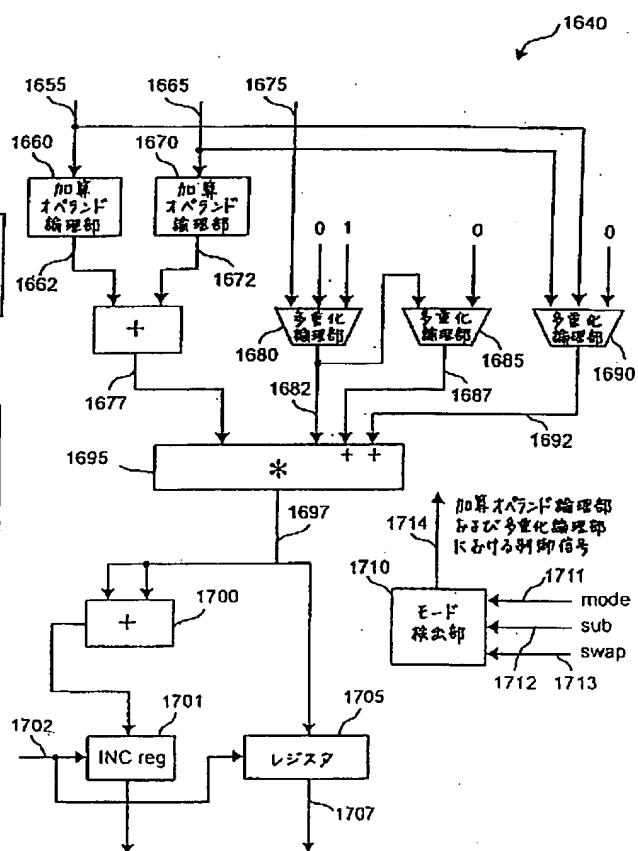
【图 126】



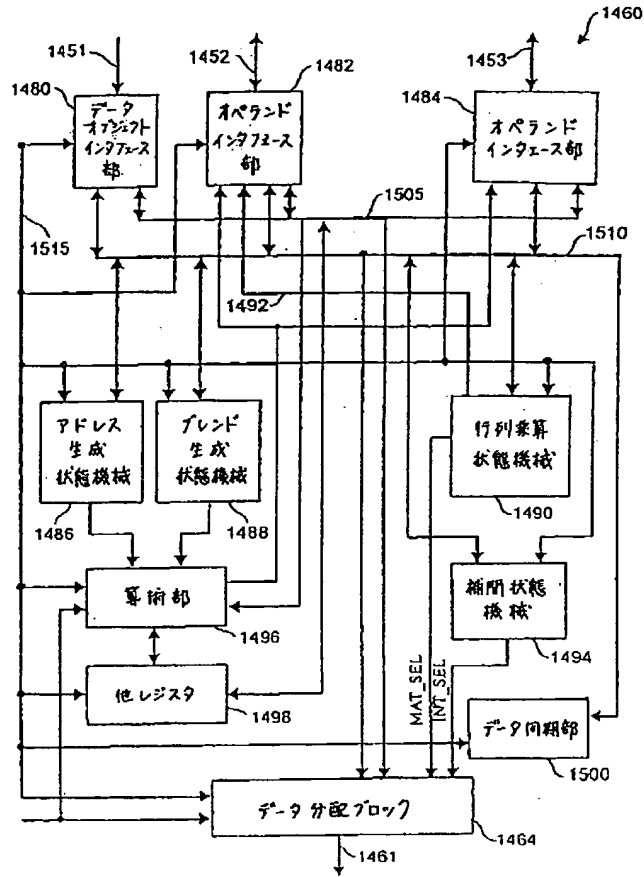
【図 129】



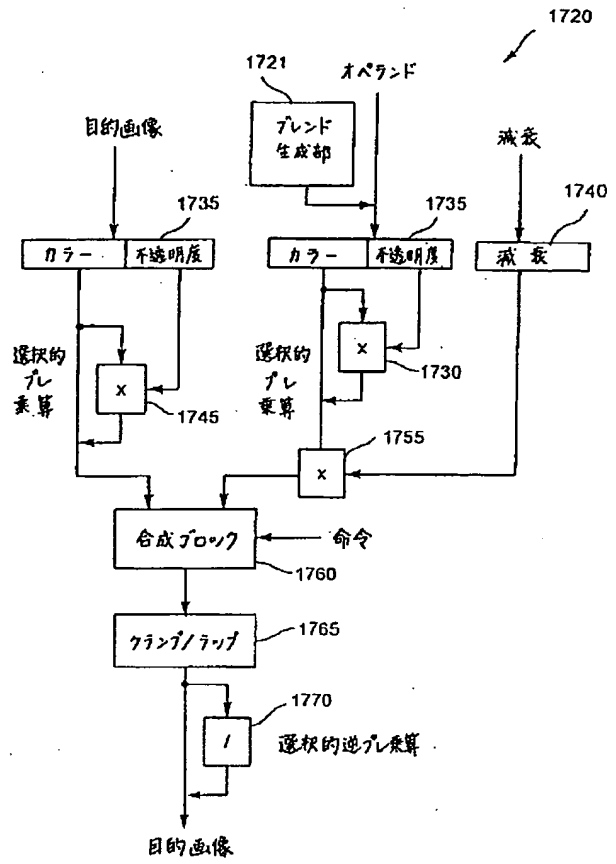
【图 133】



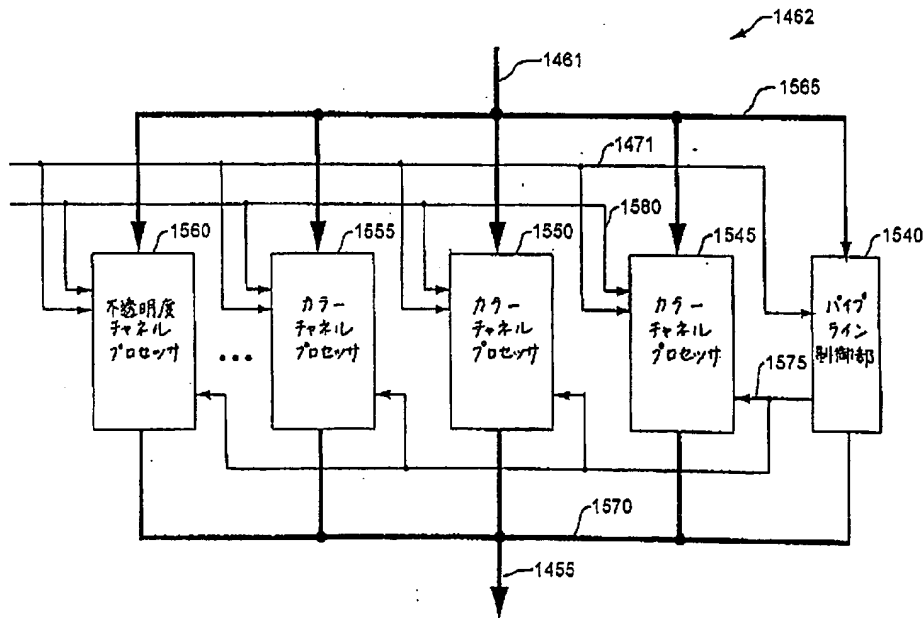
【図130】



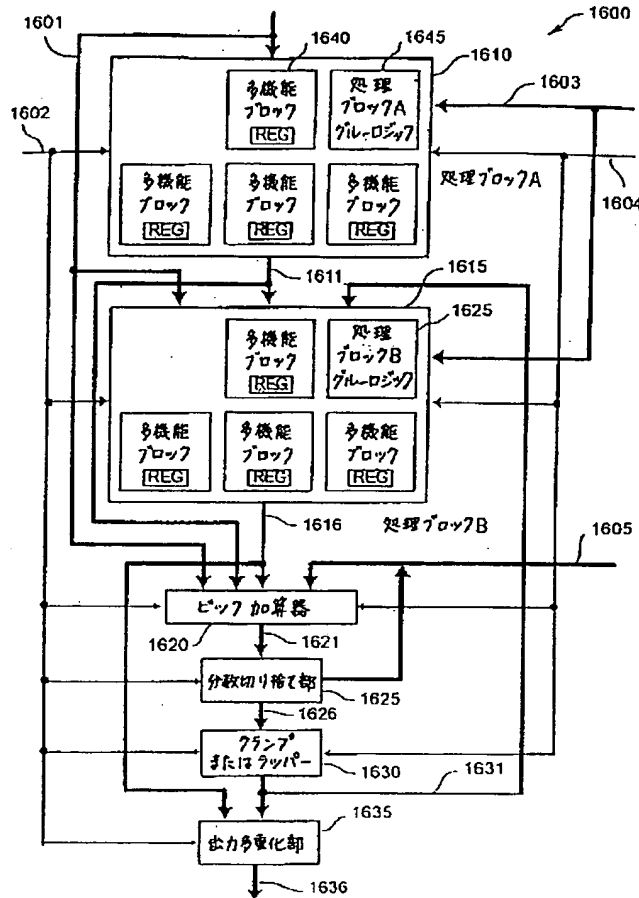
【図134】



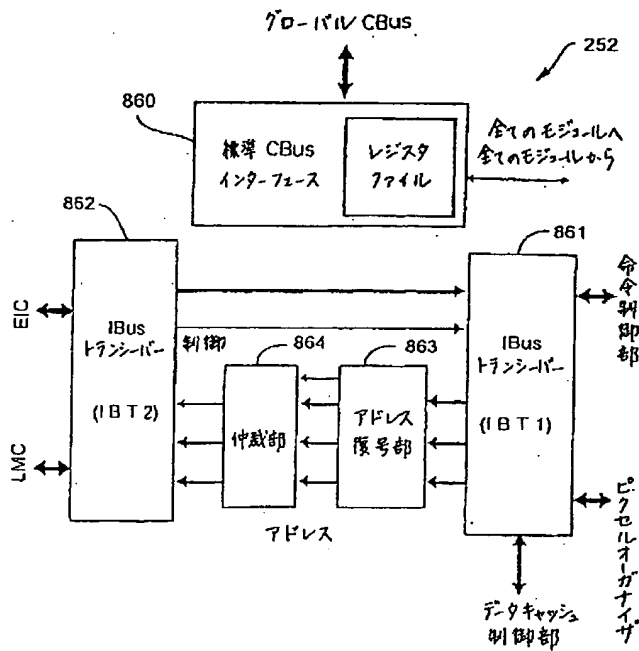
【図131】



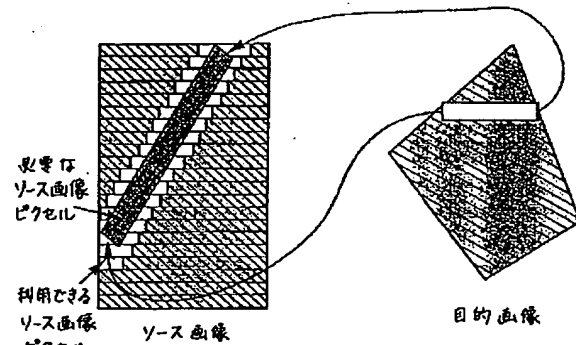
【図132】



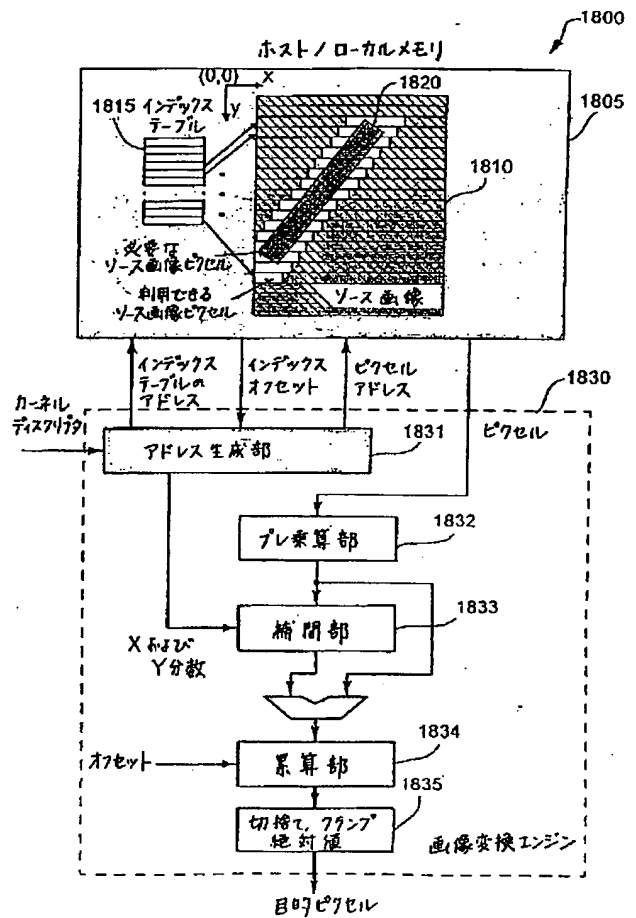
【図145】



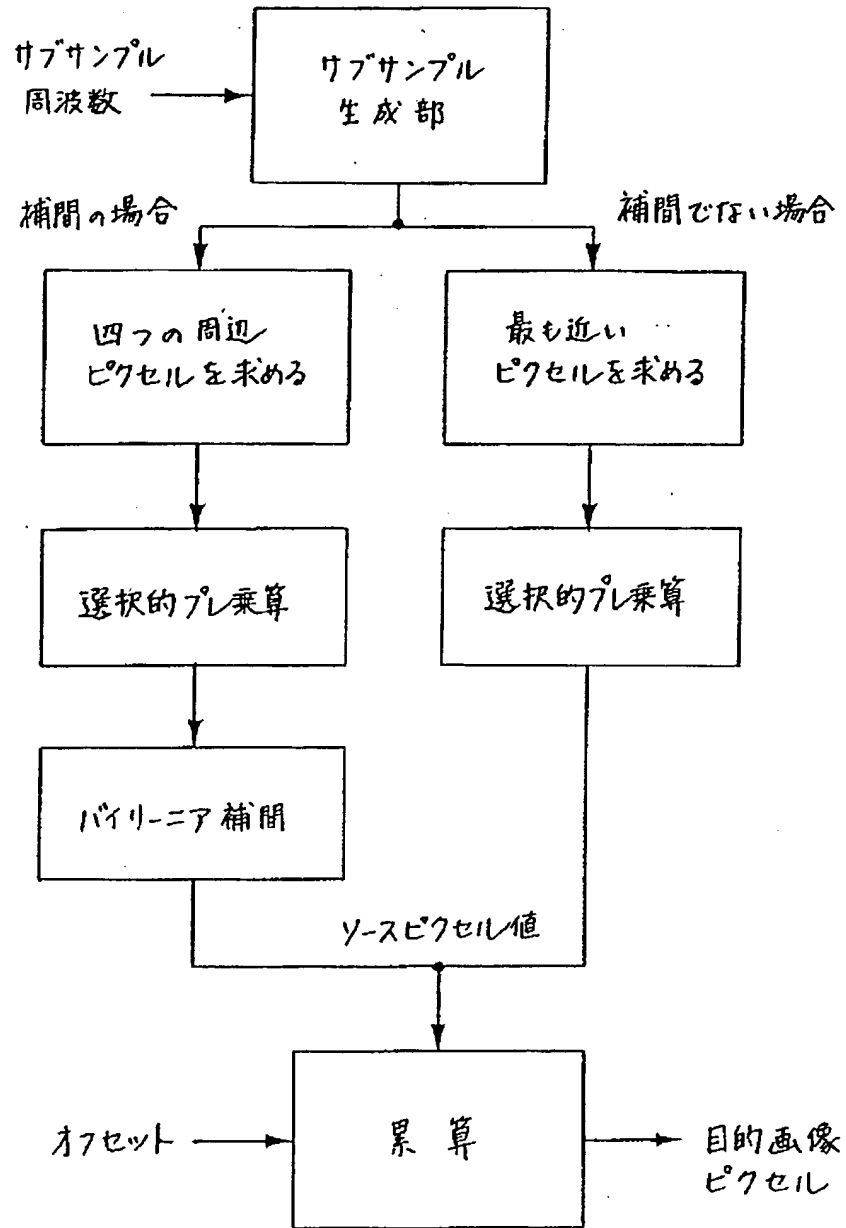
【図135】



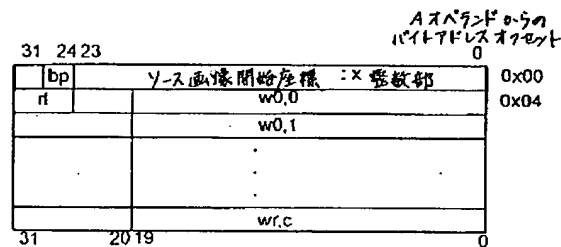
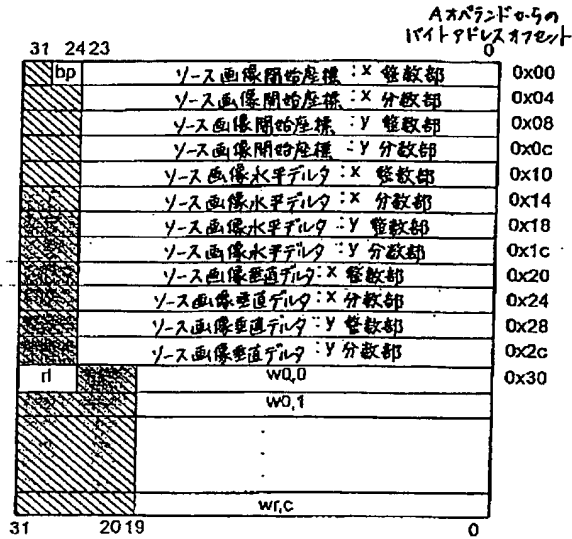
【図137】



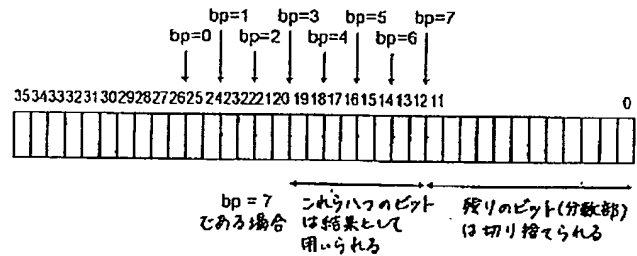
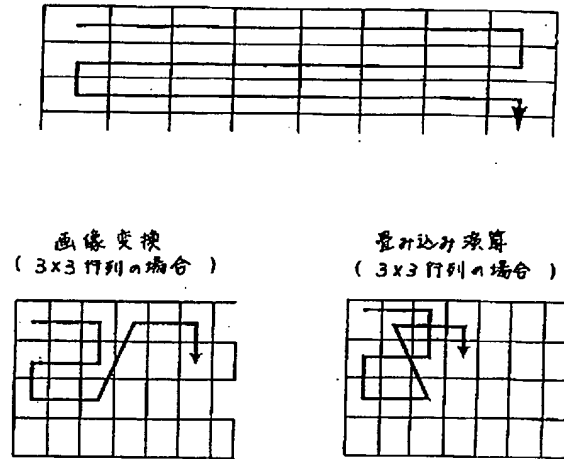
【図136】



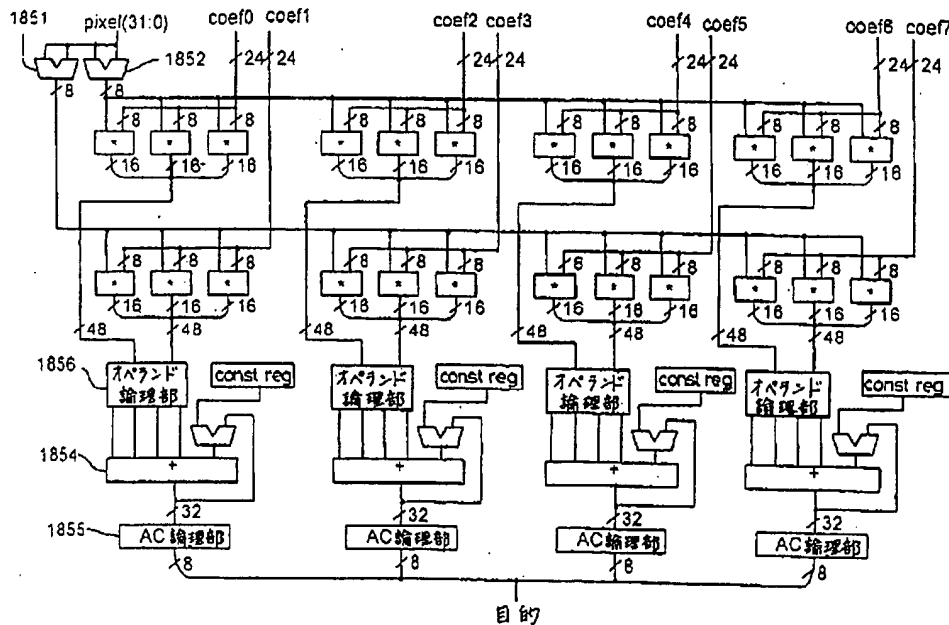
【図138】



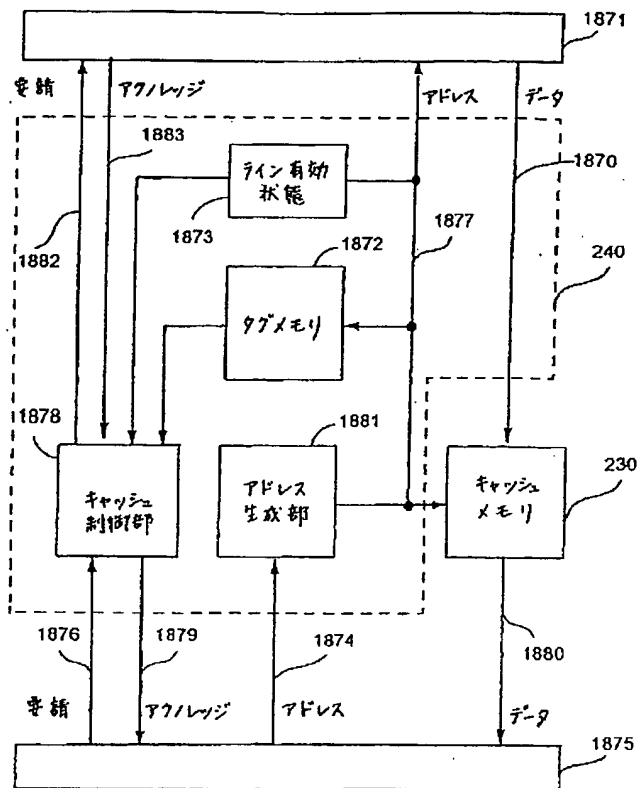
【図139】



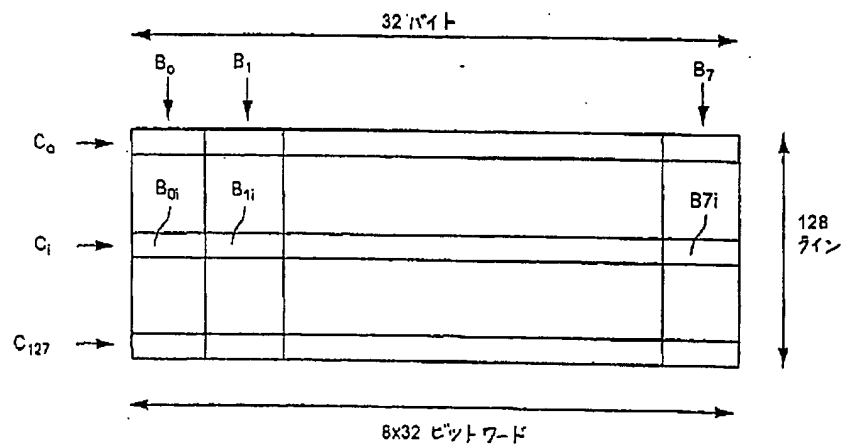
【図140】



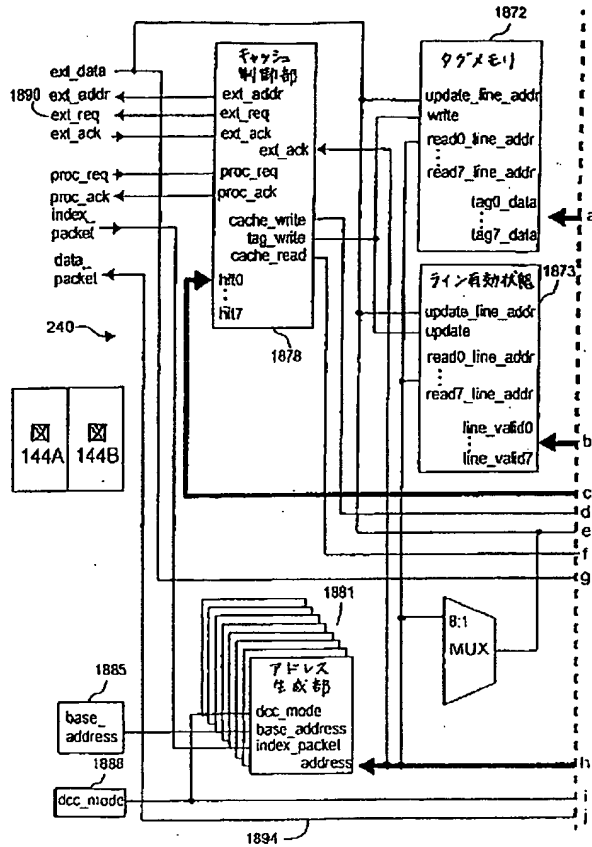
【図 141】



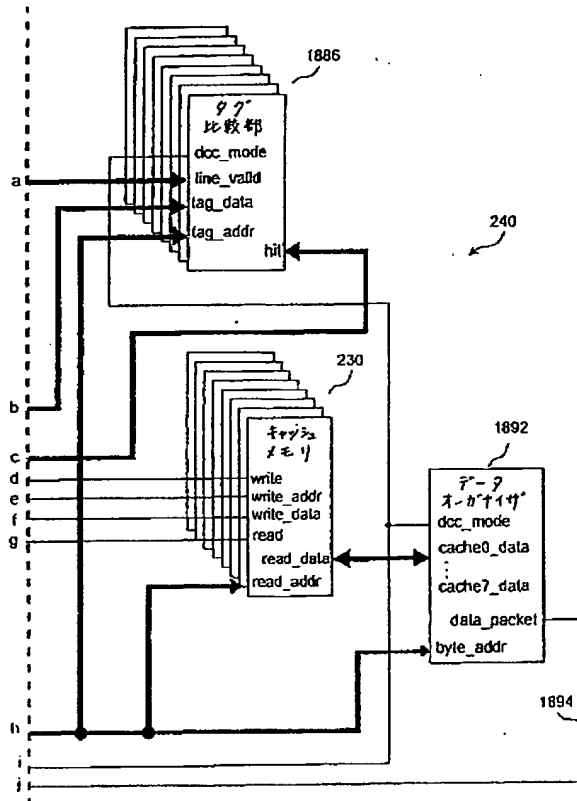
【图 142】



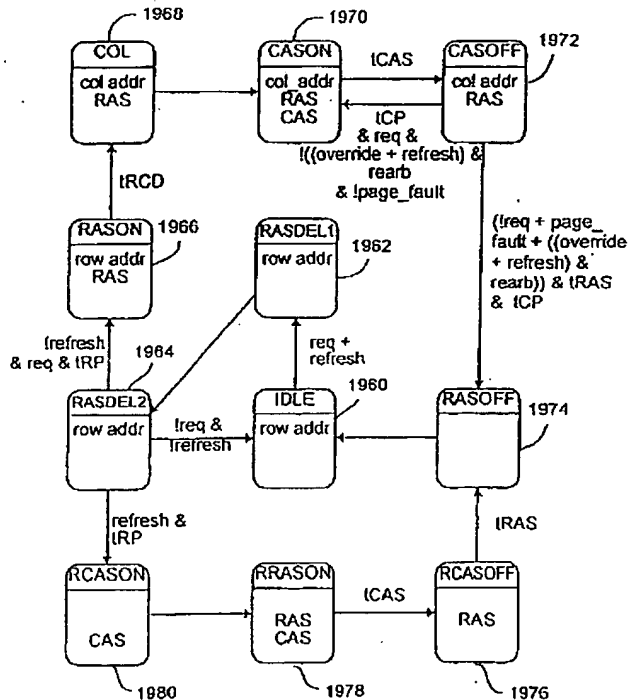
【図144A】



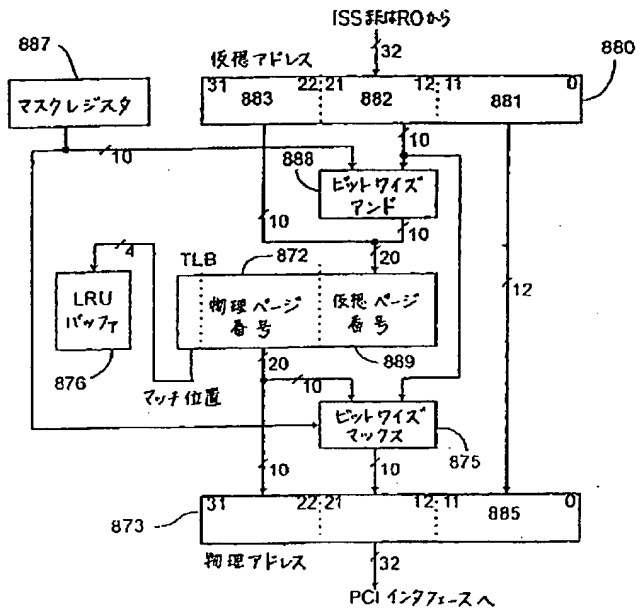
【図144B】



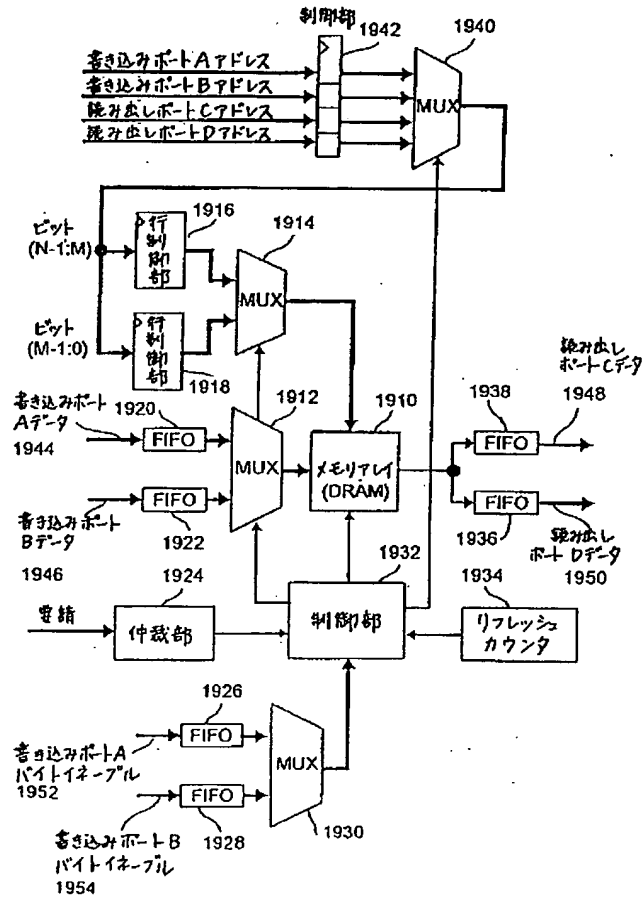
【図147】



【図152】



【図146】



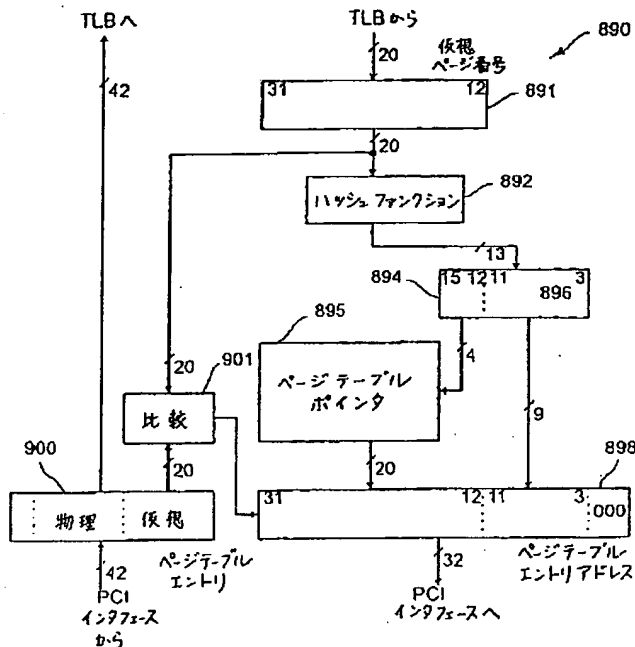
【図148】

```

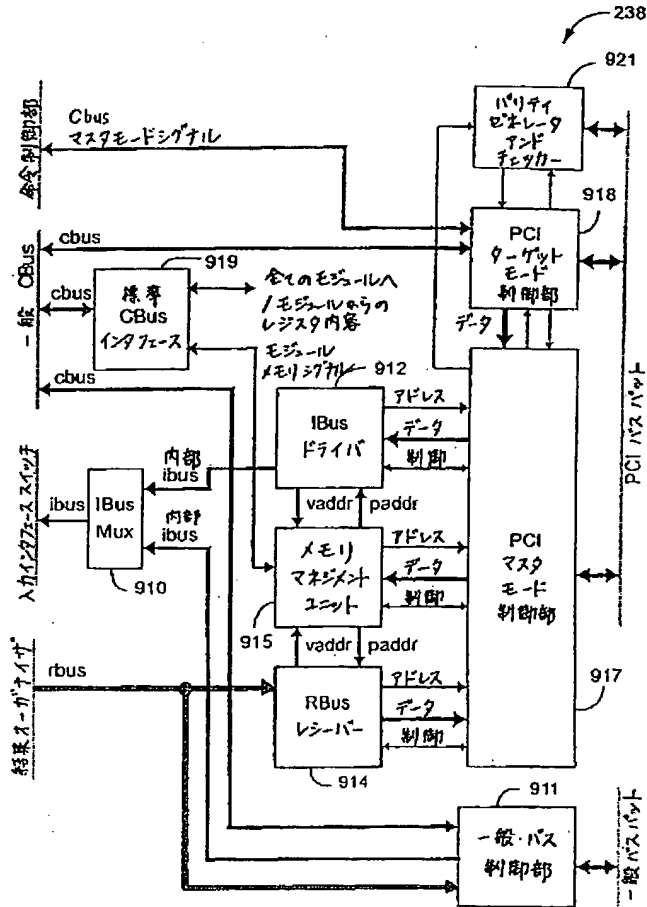
IF (AREQ) THEN
  IF (BREQ & (A>B) + CREQ & (A>C) + DREQ & (A>D)) THEN
    GRANT A
  IF (BREQ) THEN
    IF (AREQ & !(A>B) + CREQ & (B>C) + DREQ & (B>D)) THEN
      GRANT B
  IF (CREQ) THEN
    IF (AREQ & !(A>C) + BREQ & !(B>C) + DREQ & (C>D)) THEN
      GRANT C
  IF (DREQ) THEN
    IF (AREQ & !(A>D) + BREQ & !(B>D) + CREQ & !(C>D)) THEN
      GRANT D

IF (GRANT A) THEN (
  IF ((AU==BU) & (AL>BL)) THEN
    BL++
  IF ((AU==CU) & (AL>CL)) THEN
    CL++
  IF ((AU==DU) & (AL>DL)) THEN
    DL++
  AL = 0
)
ELSEIF (GRANT B) THEN (
  IF ((AU==BU) & (AL>BL)) THEN
    AL++
  IF ((BU==CU) & (BL>CL)) THEN
    CL++
  IF ((BU==DU) & (BL>DL)) THEN
    DL++
  BL = 0
)
ELSEIF (GRANT C) THEN (
  IF ((AU==CU) & (AL>CL)) THEN
    AL++
  IF ((BU==CU) & (BL>CL)) THEN
    BL++
  IF ((CU==DU) & (CL>DL)) THEN
    DL++
  CL = 0
)
ELSEIF (GRANT D) THEN (
  IF ((AU==DU) & (AL>DL)) THEN
    AL++
  IF ((BU==DU) & (BL>DL)) THEN
    BL++
  IF ((CU==DU) & (CL>DL)) THEN
    CL++
  DL = 0
)
  
```

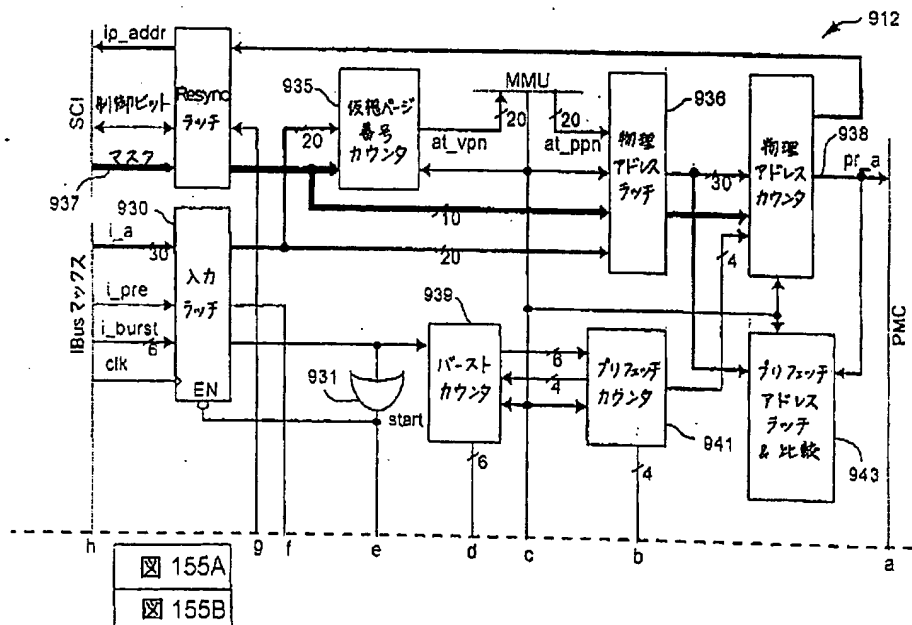
【図154】



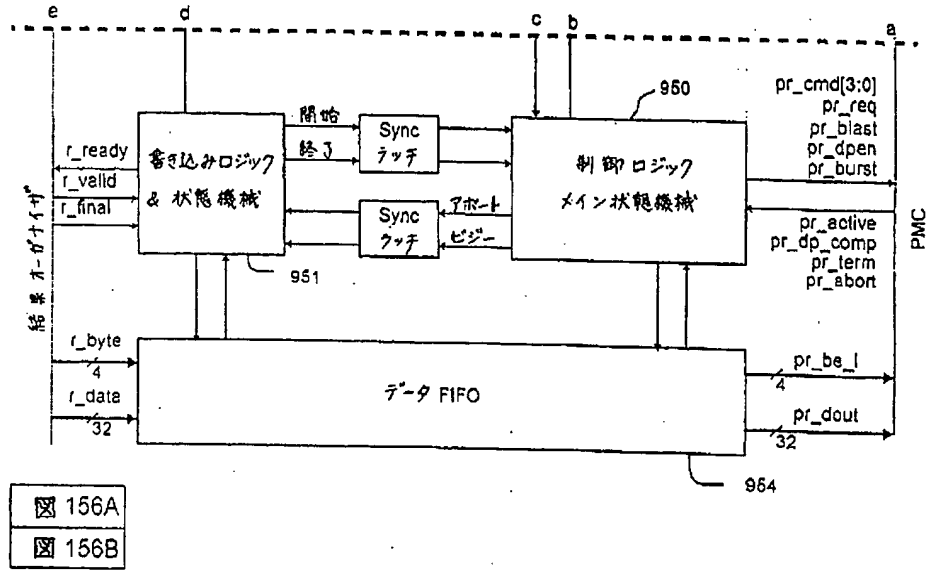
【図150】



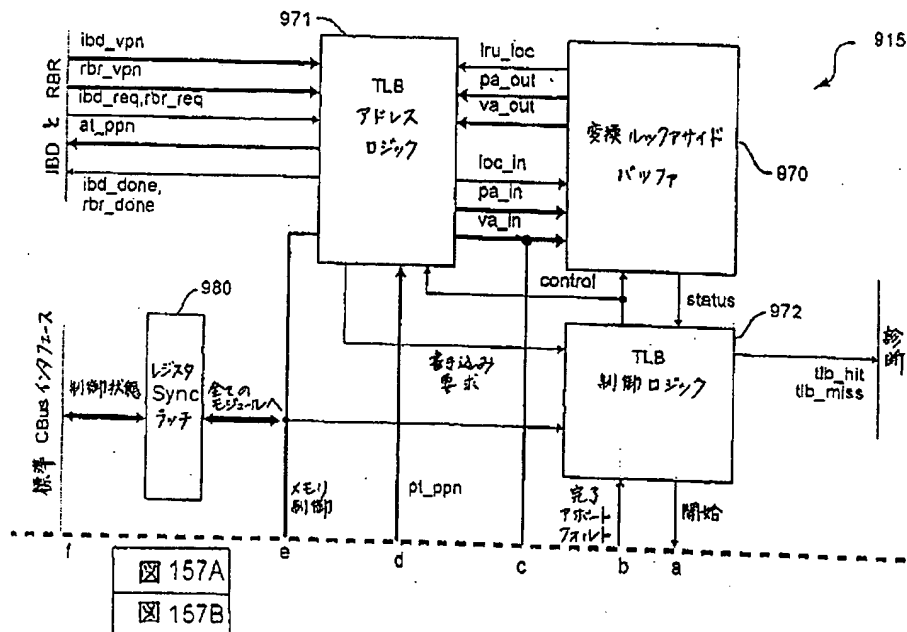
【図155A】



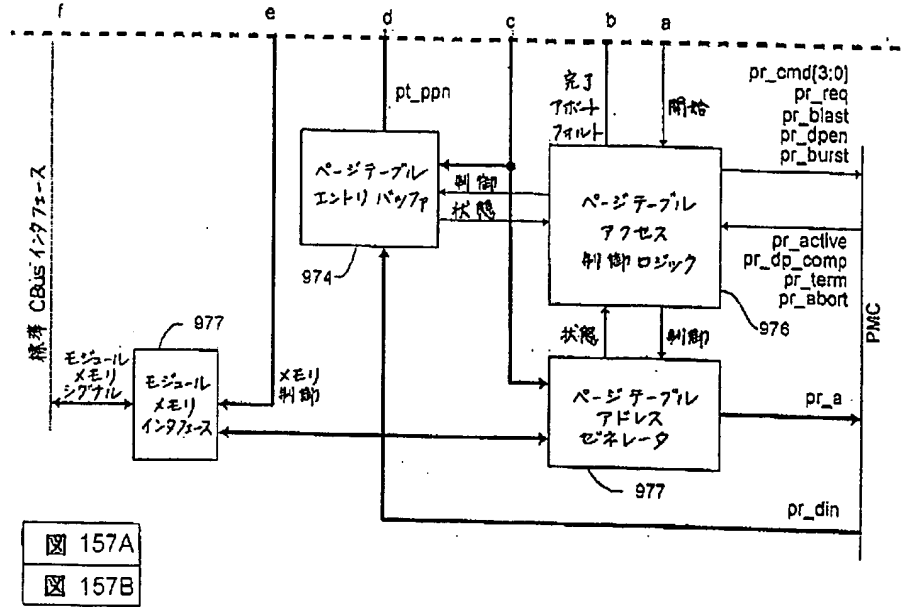
【図156B】



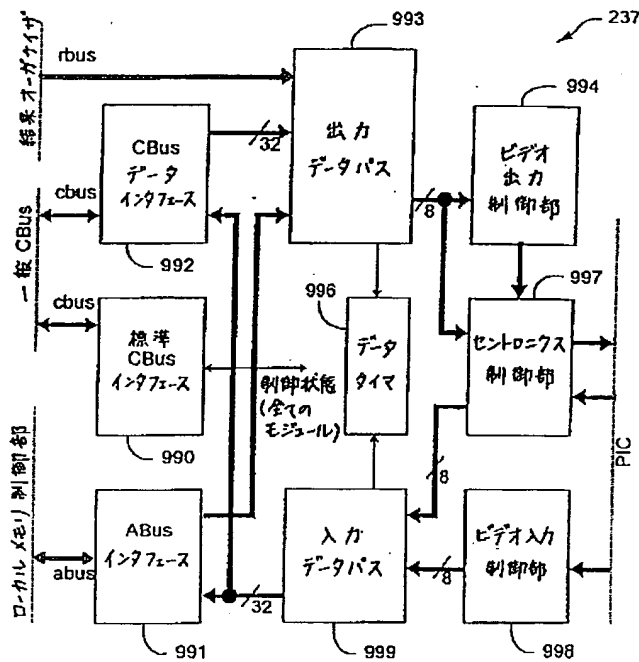
【図157A】



【図157B】



【図158】



フロントページの続き

(71)出願人 000001007

キヤノン株式会社

東京都大田区下丸子3丁目30番2号

(72)発明者 トマツ トーマス プロコブ

オーストラリア国 2150 ニューサウスウ

ェールズ州 パラマッティ, カンベル

ストリート 13ディ/15

(72)発明者 トレバー ロバート エルボーン
オーストラリア国 2094 ニューサウスウ
ェールズ州 フェアライト, ヒルトップ
クレセント 5/54

(72)発明者 マーク プルバー
オーストラリア国 2042 ニューサウスウ
ェールズ州 エンモア, トラファルガー
ストリート 15

1. TITLE OF THE INVENTION

5 COMPRESSION APPARATUS AND METHOD THEREFOR

10

15 2. CLAIMS

1. An apparatus for decoding blocks of data encoded using a plurality of variable-length code words, said blocks of data also comprising not-encoded fields of fixed length, said code words being interleaved with not-
20 encoded bit fields of variable length, said apparatus comprising:

a preprocessing logic unit for removing a plurality of not-encoded fields of fixed length and outputting said plurality of variable-length code words interleaved with said not-encoded bit fields of variable length, and outputting signals indicating positions of said plurality of not-encoded fields of fixed
25 length in the blocks of data;

means for passing said signals indicating positions to an output of the apparatus in a way synchronous with the data being decoded, such that if there is a multiplicity of variable-length coded data between said not-encoded fields of fixed length on the input of the apparatus, a multiplicity of
30 corresponding decoded data appears on the output of the apparatus between

any two signals indicating positions corresponding to said not-encoded words of fixed length on the input.

2. The apparatus according to claim 1, further comprising:

5 a first processing unit comprising a first set of barrel shifters and a first register, wherein said first processing unit processes said outputted plurality of variable-length code words interleaved with said not-encoded bit fields of variable length; and

a second processing unit comprising a second set of barrel shifters and
10 a second register, said second processing unit for processing said outputted signals indicating positions of said not-encoded words of fixed length in the blocks of data;

wherein said first and second processing units are identical, and the output of said respective barrel shifters and said units receive identical
15 control signals.

3. The apparatus according to claim 2, wherein the output of said second processing unit for processing signals indicating positions of said not-encoded words of fixed length is used to determine the size of a not-encoded
20 variable length field to be removed from the data stored in a data register for decoding purposes.

4. The apparatus according to claim 1, wherein said preprocessing logic unit for removing said not-encoded fields of fixed length
25 outputs a plurality of variable-length code words interleaved with said not-encoded bit fields of variable length as words of fixed length composed of bit fields of fixed length such that a single bit field has a corresponding tag indicating if the outputted field is passed or removed by said preprocessing unit, or passed by said preprocessing unit and following or preceding a marker
30 being a not-encoded field of fixed length.

5. The apparatus according to claim 1, wherein said blocks of data are encoded using Huffman coding.

5 6. A method of decoding blocks of data which have been encoded using a plurality of variable-length code words, said blocks of data comprising not-encoded fields of fixed length, said code words being interleaved with not-encoded bit fields of variable length, said method comprising the steps:

removing said not-encoded fields of fixed length, and outputting said
10 plurality of variable-length code words interleaved with said not-encoded bit fields of variable length, and outputting signals indicating positions of the said not-encoded words of fixed length in the blocks of data;

passing said signals indicating positions to an output in a way
synchronous with the data being decoded, such that if there is a multiplicity of
15 variable-length coded data between said not-encoded fields of fixed length being input, a multiplicity of corresponding decoded data appears at the output between any two signals indicating positions corresponding to said not-encoded words of fixed length at the input.

20 7. The method according to claim 6, further comprising the steps of:

processing said outputted plurality of variable-length code words
interleaved with said not-encoded bit fields of variable length using a first
processing unit comprising a first set of barrel shifters and a first register;
25 and

processing said outputted signals indicating positions of said not-encoded words of fixed length in the blocks of data using a second processing unit comprising a second set of barrel shifters and a second register;

wherein said first and second processing units are identical, and the output of said respective barrel shifters and said units receive identical control signals.

5 8. The method according to claim 7, further comprising the step of determining the size of a not-encoded variable length field to be removed from the data stored in a data register for decoding purposes dependent upon the output of said second processing unit for processing signals indicating positions of said not-encoded words of fixed length.

10

9. The method according to claim 6, further comprising the step of:

outputting, by said preprocessing logic unit for removing said not-encoded fields of fixed length, a plurality of variable-length code words
15 interleaved with said not-encoded bit fields of variable length as words of fixed length composed of bit fields of fixed length such that a single bit field has a corresponding tag indicating if the outputted field is passed or removed by said preprocessing unit, or passed by said preprocessing unit and following or preceding a marker being a not-encoded field of fixed length.

20

10. The method according to claim 6, wherein said blocks of data are encoded using Huffman coding.

11. A discrete cosine transform (DCT) apparatus, comprising:
25 a transpose memory means; and
an arithmetic circuit interconnected with said transpose memory means, said arithmetic circuit consisting of a combinatorial circuit for calculating a DCT without clocked storage means.

12. The DCT apparatus according to claim 11, wherein said combinatorial circuit comprises a predetermined number of stages for implementing said DCT, said stages arranged sequentially.

5 13. The DCT apparatus according to claim 11, further comprising means for multiplexing input data provided to said apparatus and data output by said transpose memory means.

14. The DCT apparatus according to claim 11, further comprising
10 means for controlling operation of said DCT apparatus.

15. An inverse discrete cosine transform (DCT) apparatus, comprising:

a transpose memory means; and
15 an arithmetic circuit interconnected with said transpose memory means, said arithmetic circuit consisting essentially of a combinatorial circuit for calculating an inverse DCT without clocked storage means.

16. The inverse DCT apparatus according to claim 15, wherein
20 said combinatorial circuit comprises a predetermined number of stages for implementing said inverse DCT, said stages being arranged sequentially.

17. The inverse DCT apparatus according to claim 15, further comprising means for multiplexing input data provided to said apparatus and
25 data output by said transpose memory means.

18. The inverse DCT apparatus according to claim 15, further comprising means for controlling operation of said DCT apparatus.

19. A method of discrete cosine transforming (DCT) data, said method comprising the steps of:

calculating a DCT of input data in accordance with a first orientation of said data using an arithmetic circuit consisting essentially of a

5 combinatorial circuit for calculating said DCT without clocked storage means;

storing said transformed input data in accordance with said first orientation in a transpose memory means interconnected with said combinatorial circuit; and

calculating a DCT of said transformed input data stored in said
10 transpose memory means in accordance with a second orientation of said data using said arithmetic circuit to provide transformed data.

20. The method according to claim 19, wherein said DCT is calculated in a predetermined number of stages, said stages arranged
15 sequentially.

21. The method according to claim 19, further comprising the step of multiplexing input data provided to said apparatus and data output by said transpose memory means.

20

22. A method of inverse discrete-cosine transforming (DCT) data, said method comprising the steps of:

calculating an inverse DCT of input coefficients in accordance with a first orientation of said coefficients using an arithmetic circuit consisting

25 essentially of a combinatorial circuit for calculating said inverse DCT without clocked storage means;

storing said inverse transformed input coefficients in accordance with said first orientation in a transpose memory means interconnected with said combinatorial circuit; and

calculating an inverse DCT of said transformed input coefficients stored in said transpose memory means in accordance with a second orientation using said arithmetic circuit to provide output inverse transformed data.

5

23. The method according to claim 22, wherein said inverse DCT is calculated in a predetermined number of stages, said stages arranged sequentially.

10

24. The method according to claim 22, further comprising the step of multiplexing input data provided to said apparatus and coefficients output by said transpose memory means.

25. A recording medium storing computer-executable program codes in which a method according to any one of claims 6 - 10 and 19 - 24 is implemented.

15

3. DETAILED DESCRIPTION OF THE INVENTION

FIELD OF THE INVENTION

The present invention relates to decoders for decoding codes of
5 variable length that are interleaved with variable-length bit fields not being
encoded, some of which are passed unchanged through the decoder. The
length of the not-encoded bit fields are equal to or greater than zero.

The present invention relates to a discrete cosine transform (DCT)
apparatus utilizing a data path, which contains no pipelining or storage
10 means, and is able to operate at high speeds.

BACKGROUND OF THE INVENTION

Generally, large amounts of data being compressed and decompressed
for numerous reasons, including transmission, storage, retrieval, and
15 processing use at some stage means of variable-length coding, such as
Huffman coding. Huffman coding was originally disclosed by D.A. Huffman
in an article "A Method for the Construction of Minimum Redundancy Codes"
Proc. IRE, 40: 1098, 1952. In many cases, variable-length codes in an
encoded bit stream are not contiguous, but are interleaved with other not-
20 encoded bit fields. The bit fields may represent control and/or formatting
information, and/or provide additional specification for encoded data including
marker headers, marker codes, stuff bytes, padding bits and additional bits in,
for example, JPEG encoded data.

Variable length encoding allocates codes of different lengths to
25 different input data according to the probability of occurrence of the input
data, so that statistically more frequent input codes are allocated shorter
codes than the less frequent codes. The less frequent input codes are
allocated longer codes. The allocation of codes may be done either statically
or adaptively. For the static case, the same output code is provided for a
30 given input datum, no matter what block of data is being processed. For the

adaptive case, output codes are assigned to input data based on a statistical analysis of a particular input block or set of blocks of data, and possibly changes from block to block (or from a set of blocks to a set of blocks).

Significant problems arise when high speed decoding of variable
 5 length codes is required. Problems particularly arise when the encoded stream contains variable-length bit fields that are not encoded are interleaved with encoded data as in, for example, the JPEG standard. Greater difficulty in fast decoding of such variable-length encoded data occurs when the length of a particular not-encoded bit field can be determined only
 10 after the preceding (encoded) datum is fully decoded, as in JPEG standard. This generally excludes direct pipelining from being incorporated into a decoder, because the position of the beginning of the next encoded datum is known only after the preceding one is fully decoded.

Existing solutions either require several steps (clock cycles) to decode
 15 a single input datum, which renders them too slow for many applications, or use iterative units to decode more than one symbol quasi-simultaneously, i.e. in one stage (clock cycle). However, additional decoding blocks often make such decoders economically unjustifiable, and further are not necessarily fast enough. This is because multiple decoding units do not work fully in parallel,
 20 since the beginning of processing in the next unit still depends on the result of processing in the previous one which determines the beginning of the input data for the next one. Thus, even if decoding of several symbols takes place in one stage (clock cycle), that stage (clock period) is relatively long and the whole decoder is too slow for many applications.

25 Therefore, a need clearly exists for a decoder that can decode variable-length codes interleaved with variable-length bit fields that are not encoded overcoming one or more of the disadvantages of conventional decoders.

Typically, a discrete cosine transform (DCT) apparatus as shown in Fig. 77 performs a full two-dimensional (2-D) transformation of a block of 8x8
 30 pixels by first performing a 1-D DCT on the rows of the 8x8 pixel block. It

then performs another 1-D DCT on the columns of the 8×8 pixel block. Such an apparatus typically consists of an input circuit 1096, an arithmetic circuit 1104, a control circuit 1098, a transpose memory circuit 1090, and an output circuit 1092.

5 The input circuit 1096 accepts 8-bit pixels from the 8×8 block. The input circuit 1096 is coupled by intermediate multiplexers 1100, 1102 to the arithmetic circuit 1104. The arithmetic circuit 1104 performs mathematical operations on either a complete row or column of the 8×8 block. The control circuit 1098 controls all the other circuits, and thus implements the DCT
10 algorithm. The output of the arithmetic circuit is coupled to the transpose memory 1090, register 1095 and output circuit 1092. The transpose memory is in turn connected to multiplexer 1100, which provides output to the next multiplexer 1102. The multiplexer 1102 also receives input from the register 1094. The transpose circuit 1090 accepts 8×8 block data in rows and
15 produces that data in columns. The output circuit 1092 provides the coefficients of the DCT performed on a 8×8 block of pixel data.

In a typical DCT apparatus, it is the speed of the arithmetic circuit 1104 that basically determines the overall speed of the apparatus, since the arithmetic circuit 1104 is the most complex.

20 The arithmetic circuit 1104 of Fig. 77 is typically implemented by breaking the arithmetic process down into several stages as described hereinafter with reference to Fig. 78. A single circuit is then built that implements each of these stages 1114, 1148, 1152, 1156 using a pool of common resources, such as adders and multipliers. Such a circuit 1104 is
25 mainly disadvantageous due to it being slower than optimal, because a single, common circuit is used to implement the various stages of circuit 1104. This includes a storage means used to store intermediate results. Since the time allocated for the clock cycle of such a circuit must be greater or equal to the time of the slowest stage of the circuit, the overall time is potentially longer
30 than the sum of all the stages.

Fig. 78 depicts a typical arithmetic data path, in accordance with the apparatus of Fig. 77, as part of a DCT with four stages. The drawing does not reflect the actual implementation, but instead reflects the functionality. Each of the four stages 1144, 1148, 1152, and 1156 is implemented using a single, reconfigurable circuit. It is reconfigured on a cycle-by-cycle basis to implement each of the four arithmetic stages 1144, 1148, 1152, and 1156 of the 1-D DCT. In this circuit, each of the four stages 1144, 1148, 1152, and 1156 uses pool of common resources (e.g. adders and multipliers) and thus minimises hardware.

However, the disadvantage of this circuit is that it is slower than optimal. The four stages 1144, 1148, 1152, and 1156 are each implemented from the same pool of adders and multipliers. The period of the clock is determined by the speed of the slowest stage, which in this example is 20 ns (for block 1144). Adding in the delay (2ns each) of the input and output multiplexers 1146 and 1154 and the delay (3ns) of the flip-flop 1150, the total time is 27 ns. Thus, the fastest this DCT implementation can run at is 27 ns.

Pipelined DCT implementations are also well known. The drawback with such implementations is that they require large amounts of hardware to implement. Whilst the present invention does not offer the same performance in terms of throughput, it offers an extremely good performance/size compromise, and good speed advantages over most of the current DCT implementations.

Therefore, a need clearly exists for an improved DCT/ inverse-DCT method and apparatus that is able to overcome one or more disadvantages of conventional techniques. In particular, a need clearly exists for a method and apparatus that is able to reduce the time taken for the main arithmetic circuit in a DCT/inverse-DCT apparatus to calculate required results, thereby improving the overall performance of the DCT or inverse DCT.

SUMMARY OF THE INVENTION

In accordance with a first aspect of the invention, there is provided an apparatus for decoding blocks of data encoded using a plurality of variable-length code words, the blocks of data also comprising not-encoded fields of fixed length, the code words being interleaved with not-encoded bit fields of
5 variable length, the apparatus comprising:

a preprocessing logic unit for removing a plurality of not-encoded fields of fixed length and outputting the plurality of variable-length code words interleaved with the not-encoded bit fields of variable length, and outputting
10 signals indicating positions of the plurality of not-encoded fields of fixed length in the blocks of data;

means for passing the signals indicating positions to an output of the apparatus in a way synchronous with the data being decoded, such that if there is a multiplicity of variable-length coded data between the not-encoded
15 fields of fixed length on the input of the apparatus, a multiplicity of corresponding decoded data appears on the output of the apparatus between any two signals indicating positions corresponding to the not-encoded words of fixed length on the input.

Preferably, the apparatus further comprises: a first processing unit
20 comprising a first set of barrel shifters and a first register, wherein the first processing unit processes the outputted plurality of variable-length code words interleaved with the not-encoded bit fields of variable length; and a second processing unit comprising a second set of barrel shifters and a second register, the second processing unit for processing the outputted signals
25 indicating positions of the not-encoded words of fixed length in the blocks of data; wherein the first and second processing units are identical, and the output of the respective barrel shifters and the units receive identical control signals. Further, the output of the second processing unit for processing signals indicating positions of the not-encoded words of fixed length may be

used to determine the size of a not-encoded variable length field to be removed from the data stored in a data register for decoding purposes.

Preferably, the preprocessing logic unit for removing the not-encoded fields of fixed length outputs a plurality of variable-length code words interleaved with the not-encoded bit fields of variable length as words of fixed length composed of bit fields of fixed length such that a single bit field has a corresponding tag indicating if the outputted field is passed or removed by the preprocessing unit, or passed by the preprocessing unit and following or preceding a marker being a not-encoded field of fixed length.

10 Preferably, the blocks of data are encoded using Huffman coding.

In accordance with a second aspect of the invention, there is provided a method of decoding blocks of data which have been encoded using a plurality of variable-length code words, the blocks of data comprising not-encoded fields of fixed length, the code words being interleaved with not-encoded bit fields of variable length, the method comprising the steps:

removing the not-encoded fields of fixed length, and outputting the plurality of variable-length code words interleaved with the not-encoded bit fields of variable length, and outputting signals indicating positions of the not-encoded words of fixed length in the blocks of data;

20 passing the signals indicating positions to an output in a way synchronous with the data being decoded, such that if there is a multiplicity of variable-length coded data between the not-encoded fields of fixed length being input, a multiplicity of corresponding decoded data appears at the output between any two signals indicating positions corresponding to the not-encoded words of fixed length at the input.

Preferably, the method further comprises the steps of: processing the outputted plurality of variable-length code words interleaved with the not-encoded bit fields of variable length using a first processing unit comprising a first set of barrel shifters and a first register; and processing the outputted signals indicating positions of the not-encoded words of fixed length in the

blocks of data using a second processing unit comprising a second set of barrel shifters and a second register; wherein the first and second processing units are identical, and the output of the respective barrel shifters and the units receive identical control signals. Further, the method may comprise the step
5 of determining the size of a not-encoded variable length field to be removed from the data stored in a data register for decoding purposes dependent upon the output of the second processing unit for processing signals indicating positions of the not-encoded words of fixed length.

Preferably, the method may further comprise the step of outputting,
10 by the preprocessing logic unit for removing the not-encoded fields of fixed length, a plurality of variable-length code words interleaved with the not-encoded bit fields of variable length as words of fixed length composed of bit fields of fixed length such that a single bit field has a corresponding tag indicating if the outputted field is passed or removed by the preprocessing
15 unit, or passed by the preprocessing unit and following or preceding a marker being a not-encoded field of fixed length.

Preferably, the blocks of data are encoded using Huffman coding.

In the following detailed description, the reader's attention is directed, in particular, to Figs. 82 to 91 and their associated description without
20 intending to detract from the disclosure of the remainder of the description.

In accordance with a third aspect of the invention, there is provided a discrete cosine transform (DCT) apparatus, comprising: a transpose memory means; and an arithmetic circuit interconnected with the transpose memory
25 means, the arithmetic circuit consisting of a combinatorial circuit for calculating a DCT without clocked storage means.

Preferably, the combinatorial circuit comprises a predetermined number of stages for implementing the DCT, the stages arranged sequentially.

Preferably, the DCT apparatus further comprises means for multiplexing input data provided to the apparatus and data output by the transpose memory means. It may also comprise means for controlling operation of the DCT apparatus.

5 In accordance with a fourth aspect of the invention, there is provided an inverse discrete cosine transform (DCT) apparatus, comprising: a transpose memory means; and an arithmetic circuit interconnected with the transpose memory means, the arithmetic circuit consisting essentially of a combinatorial circuit for calculating an inverse DCT without clocked storage
10 means.

In accordance with a third aspect of the invention, there is provided a method of discrete cosine transforming (DCT) data, the method comprising the steps of:

calculating a DCT of input data in accordance with a first orientation
15 of the data using an arithmetic circuit consisting essentially of a combinatorial circuit for calculating the DCT without clocked storage means;

storing the transformed input data in accordance with the first orientation in a transpose memory means interconnected with the combinatorial circuit; and

20 calculating a DCT of the transformed input data stored in the transpose memory means in accordance with a second orientation of the data using the arithmetic circuit to provide transformed data.

Preferably, the DCT is calculated in a predetermined number of stages, the stages arranged sequentially.

25 The method may also comprise the step of multiplexing input data provided to the apparatus and data output by the transpose memory means.

In accordance with a fourth aspect of the invention, there is provided a method of inverse discrete-cosine transforming (DCT) data, the method comprising the steps of:

calculating an inverse DCT of input coefficients in accordance with a first orientation of the coefficients using an arithmetic circuit consisting essentially of a combinatorial circuit for calculating the inverse DCT without clocked storage means;

5 storing the inverse transformed input coefficients in accordance with the first orientation in a transpose memory means interconnected with the combinatorial circuit; and

calculating an inverse DCT of the transformed input coefficients stored in the transpose memory means in accordance with a second
10 orientation using the arithmetic circuit to provide output inverse transformed data.

In the following detailed description, the reader's attention is directed, in particular, to Figs. 79, 80 and 81 and their associated description without intending to detract from the disclosure of the remainder of the description.

15

PREFERRED EMBODIMENTS

TABLE OF CONTENTS

20	1.0	Brief Description of the Drawings
	2.0	List of Tables
	3.0	Description of the Preferred and Other Embodiments
	3.1	General Arrangement of Plural Stream Architecture
	3.2	Host/Co-processor Queuing
25	3.3	Register Description of Co-processor
	3.4	Format of Plural Streams
	3.5	Determine Current Active Stream
	3.6	Fetch Instruction of Current Active Stream
	3.7	Decode and Execute Instruction
30	3.8	Update Registers of Instruction Controller

	3.9	Semantics of the Register Access Semaphore
	3.10	Instruction Contrroller
	3.11	Description of a Modules Local Register File
	3.12	Register Read/Write Handling
5	3.13	Memory Area Read/Write Handling
	3.14	CBus Structure
	3.15	Co-processor Data Types and Data Manipulation
	3.16	Data Normalization Circuit
	3.17	Image Processing Operations of Accelator Card
10	3.17.1	Compositing
	3.17.2	Color Space Conversion Instructions
	a.	Single Output General Color Space (SOGCS) Conversion Mode
	b.	Multiple Output General Color Space
15	Mode	
	3.17.3	JPEG Coding/Decoding
	a.	Encoding
	b.	Decoding
	3.17.4	Table Indexing
20	3.17.5	Data Coding Instructions
	3.17.6	A Fast DCT Apparatus
	3.17.7	Huffman Decoder
	3.17.8	Image Transformation Instructions
	3.17.9	Convolution Instructions
25	3.17.10	Matrix Multiplication
	3.17.11	Halftoning
	3.17.12	Hierarchial Image Format Decompression
	3.17.13	Memory Copy Instructions
	a.	General purpose data movement
30	instructions	

b. Local DMA instructions

3.17.14 Flow Control Instructions

3.18 Modules of the Accelerator Card

3.18.1 Pixel Organizer

5 3.18.2 MUV Buffer

3.18.3 Result Organizer

3.18.4 Operand Organizers B and C

3.18.5 Main Data Path Unit

3.18.6 Data Cache Controller and Cache

10 a. Normal Cache Mode

b. The Single Output General Color Space
Conversion Modec. Multiple Output General Color Space
Conversion Mode

15 d. JPEG Encoding Mode

e. Slow JPEG Decoding Mode

f. Matrix Multiplication Mode

g. Disabled Mode

h. Invalidate Mode

20 3.18.7 Input Interface Switch

3.18.8 Local Memory Controller

3.18.9 Miscellaneous Module

3.18.10 External Interface Controller

3.18.11 Peripheral Interface Controller

25

APPENDIX A - Microprogramming

APPENDIX B - Register tables

2.0 List of Tables

	Table 1: Register Description
	Table 2: Opcode Description
5	Table 3: Operand Types
	Table 4: Operand Descriptors
	Table 5: Module Setup Order
	Table 6: CBus Signal Definition
	Table 7: CBus Transaction Types
10	Table 8: Data Manipulation Register Format
	Table 9: Expected Data Types
	Table 10: Symbol Explanation
	Table 11: Compositing Operations
	Table 12: Address Composition for SOGCS Mode
15	Table 12A: Instruction Encoding for Color Space Conversion
	Table 13: Minor Opcode Encoding for Color Conversion Instructions
	Table 14: Huffman and Quantization Tables as stored in Data Cache
	Table 15: Fetch Address
	Table 16: Tables Used by the Huffman Encoder
20	Table 17: Bank Address for Huffman and Quantization Tables
	Table 18: Instruction Word - Minor Opcode Fields
	Table 19: Instruction Word - Minor Opcode Fields
	Table 20: Instruction Operand and Results Word
	Table 21: Instruction Word
25	Table 22: Instruction Operand and Results Word
	Table 23: Instruction Word
	Table 24: Instruction Operand and Results Word
	Table 25: Instruction Word - Minor Opcode Fields
	Table 26: Instruction Word - Minor Opcode Fields
30	Table 27: Fraction Table

3.0 Description of the Preferred and Other Embodiments

In the preferred embodiment, a substantial advantage is gained in hardware rasterization by means of utilization of two independent instruction streams by a hardware accelerator. Hence, while the first instruction stream can be preparing a current page for printing, a subsequent instruction stream can be preparing the next page for printing. A high utilization of hardware resources is available especially where the hardware accelerator is able to work at a speed substantially faster than the speed of the output device.

The preferred embodiment describes an arrangement utilising two instruction streams. However, arrangements having further instruction streams can be provided where the hardware trade-offs dictate that substantial advantages can be obtained through the utilization of further streams.

The utilization of two streams allows the hardware resources of the raster image co-processor to be kept fully engaged in preparing subsequent pages or bands, strips, etc., depending on the output printing device while a present page, band, etc is being forwarded to a print device.

3.1 General Arrangement of Plural Stream Architecture

In Fig. 1 there is schematically illustrated a computer hardware arrangement 201 which constitutes the preferred embodiment. The arrangement 201 includes a standard host computer system which takes the form of a host CPU 202 interconnected to its own memory store (RAM) 203 via a bridge 204. The host computer system provides all the normal facilities of a computer system including operating systems programs, applications, display of information, etc. The host computer system is connected to a standard PCI bus 206 via a PCI bus interface 207. The PCI standard is a well known industry standard and most computer systems sold today, particularly those running Microsoft Windows (trade mark) operating

systems, normally come equipped with a PCI bus 206. The PCI bus 206 allows the arrangement 201 to be expanded by means of the addition of one or more PCI cards, eg. 209, each of which contain a further PCI bus interface 210 and other devices 211 and local memory 212 for utilization in the arrangement 201.

In the preferred embodiment, there is provided a raster image accelerator card 220 to assist in the speeding up of graphical operations expressed in a page description language. The raster image accelerator card 220 (also having a PCI bus interface 221) is designed to operate in a loosely coupled, shared memory manner with the host CPU 202 in the same manner as other PCI cards 209. It is possible to add further image accelerator cards 220 to the host computer system as required. The raster image accelerator card is designed to accelerate those operations that form the bulk of the execution complexity in raster image processing operations. These can include:

- (a) Composition
- (b) Generalized Color Space Conversion
- (c) JPEG compression and decompression
- (d) Huffman, run length and predictive coding and decoding
- (e) Hierarchical image (Trade Mark) decompression
- (f) Generalized affine image transformations
- (g) Small kernel convolutions
- (h) Matrix multiplication
- (i) Halftoning
- (j) Bulk arithmetic and memory copy operations

The raster image accelerator card 220 further includes its own local memory 223 connected to a raster image co-processor 224 which operates the raster image accelerator card 220 generally under instruction from the host CPU 202. The co-processor 224 is preferably constructed as an Application Specific Integrated Circuit (ASIC) chip. The raster image co-processor 224

includes the ability to control at least one printer device 226 as required via a peripheral interface 225. The image accelerator card 220 may also control any input/output device, including scanners. Additionally, there is provided on the accelerator card 220 a generic external interface 227 connected with the
 5 raster image co-processor 224 for its monitoring and testing.

In operation, the host CPU 202 sends, via PCI bus 206, a series of instructions and data for the creation of images by the raster image co-processor 224. The data can be stored in the local memory 223 in addition to a cache 230 in the raster image co-processor 224 or in registers 229 also
 10 located in the co-processor 224.

Turning now to Fig. 2, there is illustrated, in more detail, the raster image co-processor 224. The co-processor 224 is responsible for the acceleration of the aforementioned operations and consists of a number of components generally under the control of an instruction controller 235.
 15 Turning first to the co-processor's communication with the outside world, there is provided a local memory controller 236 for communications with the local memory 223 of Fig. 1. A peripheral interface controller 237 is also provided for the communication with printer devices utilising standard formats such as the Centronics interface standard format or other video
 20 interface formats. The peripheral interface controller 237 is interconnected with the local memory controller 236. Both the local memory controller 236 and the external interface controller 238 are connected with an input interface switch 252 which is in turn connected to the instruction controller 235. The input interface switch 252 is also connected to a pixel organizer 246 and a
 25 data cache controller 240. The input interface switch 252 is provided for switching data from the external interface controller 238 and local memory controller 236 to the instruction controller 235, the data cache controller 240 and the pixel organizer 246 as required.

For communications with the PCI bus 206 of Fig. 1 the external
 30 interface controller 238 is provided in the raster image co-processor 224 and is

connected to the instruction controller 235. There is also provided a miscellaneous module 239 which is also connected to the instruction controller 235 and which deals with interactions with the co-processor 224 for purposes of test diagnostics and the provision of clocking and global signals.

5 The data cache 230 operates under the control of the data cache controller 240 with which it is interconnected. The data cache 230 is utilized in various ways, primarily to store recently used values that are likely to be subsequently utilized by the co-processor 224. The aforementioned acceleration operations are carried out on plural streams of data primarily by
10 a JPEG coder/decoder 241 and a main data path unit 242. The units 241, 242 are connected in parallel arrangement to all of the pixel organizer 246 and two operand organizers 247, 248. The processed streams from units 241, 242 are forwarded to a results organizer 249 for processing and reformatting where required. Often, it is desirable to store intermediate results close at
15 hand. To this end, in addition to the data cache 230, a multi-used value buffer 250 is provided, interconnected between the pixel organizer 246 and the result organizer 249, for the storage of intermediate data. The result organizer 249 outputs to the external interface controller 238, the local memory controller 236 and the peripheral interface controller 237 as required.

20 As indicated by broken lines in Fig. 2, a further (third) data path unit 243 can, if required be connected "in parallel" with the two other data paths in the form of JPEG coder/decoder 241 and the main data path unit 242. The extension to 4 or more data paths is achieved in the same way. Although the paths are "parallel" connected, they do not operate in parallel. Instead only
25 one path at a time operates.

 The overall ASIC design of Fig. 2 has been developed in the following manner. Firstly, in printing pages it is necessary that there not be even small or transient artefacts. This is because whilst in video signal creation for example, such small errors if present may not be apparent to the human
30 eye (and hence be unobservable), in printing any small artefact appears

permanently on the printed page and can sometimes be glaringly obvious. Further, any delay in the signal reaching the printer can be equally disastrous resulting in white, unprinted areas on a page as the page continues to move through the printer. It is therefore necessary to provide results of very high
5 quality, very quickly and this is best achieved by a hardware rather than a software solution.

Secondly, if one lists all the various operational steps (algorithms) required to be carried out for the printing process and provides an equivalent item of hardware for each step, the total amount of hardware becomes
10 enormous and prohibitively expensive. Also the speed at which the hardware can operate is substantially limited by the rate at which the data necessary for, and produced by, the calculations can be fetched and despatched respectively. That is, there is a speed limitation produced by the limited bandwidth of the interfaces.

15 However, overall ASIC design is based upon a surprising realization that if the enormous amount of hardware is represented schematically then various parts of the total hardware required can be identified as being (a) duplicated and (b) not operating all the time. This is particularly the case in respect of the overhead involved in presenting the data prior to its calculation.

20 Therefore various steps were taken to reach the desired state of reducing the amount of hardware whilst keeping all parts of the hardware as active as possible. The first step was the realization that in image manipulation often repetitive calculations of the same basic type were required to be carried out. Thus if the data were streamed in some way, a
25 calculating unit could be configured to carry out a specific type of calculation, a long stream of data processed and then the calculating unit could be reconfigured for the next type of calculation step required. If the data streams were reasonably long, then the time required for reconfiguration would be negligible compared to the total calculation time and thus
30 throughput would be enhanced.

In addition, the provision of plural data processing paths means that in the event that one path is being reconfigured whilst the other path is being used, then there is substantially no loss of calculating time due to the necessary reconfiguration. This applies where the main data path unit 242 carries out a more general calculation and the other data path(s) carry out more specialized calculation such as JPEC coding and decoding as in unit 241 or, if additional unit 243 is provided, it can provide entropy and/or Huffman coding/decoding.

Further, whilst the calculations were proceeding, the fetching and presenting of data to the calculating unit can be proceeding. This process can be further speeded up, and hardware resources better utilized, if the various types of data are standardized or normalized in some way. Thus the total overhead involved in fetching and despatching data can be reduced.

Importantly, as noted previously, the co-processor 224 operates under the control of host CPU 202 (Fig. 1). In this respect, the instruction controller 235 is responsible for the overall control of the co-processor 224. The instruction controller 235 operates the co-processor 224 by means of utilising a control bus 231, hereinafter known as the CBus. The CBus 231 is connected to each of the modules 236-250 inclusive to set registers (231 of Fig. 1) within each module so as to achieve overall operation of the co-processor 224. In order not to overly complicate Fig. 2, the interconnection of the control bus 231 to each of the modules 236-250 is omitted from Fig. 2.

Turning now to Fig. 3, there is illustrated a schematic layout 260 of the available module registers. The layout 260 includes registers 261 dedicated to the overall control of the co-processor 224 and its instruction controller 235. The co-processor modules 236-250 include similar registers 262.

3.2 Host/Co-processor Queuing

With the above architecture in mind, it is clear that there is a need to adequately provide for cooperation between the host processor 202 and the

image co-processor 224. However, the solution to this problem is general and not restricted to the specific above described architecture and therefore will be described hereafter with reference to a more general computing hardware environment.

5 Modern computer systems typically require some method of memory management to provide for dynamic memory allocation. In the case of a system with one or more co-processors, some method is necessary to synchronize between the dynamic allocation of memory and the use of that memory by a co-processor.

10 Typically a computer hardware configuration has both a CPU and a specialized co-processor, each sharing a bank of memory. In such a system, the CPU is the only entity in the system capable of allocating memory dynamically. Once allocated by the CPU for use by the co-processor, this memory can be used freely by the co-processor until it is no longer required, at
15 which point it is available to be freed by the CPU. This implies that some form of synchronization is necessary between the CPU and the co-processor in order to ensure that the memory is released only after the co-processor is finished using it. There are several possible solutions to this problem but each has undesirable performance implications.

20 The use of statically allocated memory avoids the need for synchronization, but prevents the system from adjusting its memory resource usage dynamically. Similarly, having the CPU block and wait until the co-processor has finished performing each operation is possible, but this substantially reduces parallelism and hence reduces overall system
25 performance. The use of interrupts to indicate completion of operations by the co-processor is also possible but imposes significant processing overhead if co-processor throughput is very high.

 In addition to the need for high performance, such a system also has to deal with dynamic memory shortages gracefully. Most computer systems
30 allow a wide range of memory size configurations. It is important that those

systems with large amounts of memory available make full use of their available resources to maximize performance. Similarly those systems with minimal memory size configurations should still perform adequately to be useable and, at the very least, should degrade gracefully in the face of a
 5 memory shortage.

To overcome these problems, a synchronization mechanism is necessary which will maximize system performance while also allowing co-processor memory usage to adjust dynamically to both the capacity of the system and the complexity of the operation being performed.

10 In general, the preferred arrangement for synchronising the (host) CPU and the co-processor is illustrated in Fig. 4 where the reference numerals used are those already utilized in the previous description of Fig. 1.

Thus in Fig. 108, the CPU 202 is responsible for all memory management in the system. It allocates memory 203 both for its own uses,
 15 and for use by the co-processor 224. The co-processor 224 has its own graphics-specific instruction set, and is capable of executing instructions 1022 from the memory 203 which is shared with the host processor 202. Each of these instructions can also write results 1024 back to the shared memory 203, and can read operands 1023 from the memory 203 as well. The amount of
 20 memory 203 required to store operands 1023 and results 1024 of co-processor instructions varies according to the complexity and type of the particular operation.

The CPU 202 is also responsible for generating the instructions 1022 executed by the co-processor 224. To maximize the degree of parallelism
 25 between the CPU 202 and the co-processor 224, instructions generated by the CPU 202 are queued as indicated at 1022 for execution by the co-processor 224. Each instruction in the queue 1022 can reference operands 1023 and results 1024 in the shared memory 203, which has been allocated by the host CPU 202 for use by the co-processor 224.

The method utilizes an interconnected instruction generator 1030, memory manager 1031 and queue manager 1032, as shown in Fig. 5. All these modules execute in a single process on the host CPU 202.

Instructions for execution by the co-processor 224 are generated by the
 5 instruction generator 1030, which uses the services of the memory manager 1031 to allocate space for the operands 1023 and results 1024 of the instructions being generated. The instruction generator 1030 also uses the services of the queue manager 1032 to queue the instructions for execution by the co-processor 224.

10 Once each instruction has been executed by the co-processor 224, the CPU 202 can free the memory which was allocated by the memory manager 1031 for use by the operands of that instruction. The result of one instruction can also become an operand for a subsequent instruction, after which its memory can also be freed by the CPU. Rather than fielding an
 15 interrupt, and freeing such memory as soon as the co-processor 224 has finished with it, the system frees the resources needed by each instruction via a cleanup function which runs at some stage after the co-processor 224 has completed the instruction. The exact time at which these cleanups occur depends on the interaction between the memory manager 1031 and the queue
 20 manager 1032, and allows the system to adapt dynamically according to the amount of system memory available and the amount of memory required by each co-processor instruction.

Fig. 6 schematically illustrates the implementation of the co-processor instruction queue 1022. Instructions are inserted into a pending
 25 instruction queue 1040 by the host CPU 202, and are read by the co-processor 224 for execution. After execution by the co-processor 224, the instructions remain on a cleanup queue 1041, so that the CPU 202 can release the resources that the instructions required after the co-processor 224 has finished executing them.

The instruction queue 1022 itself can be implemented as a fixed or dynamically sized circular buffer. The instruction queue 1022 decouples the generation of instructions by the CPU 202 from their execution by the co-processor 224.

5 Operand and result memory for each instruction is allocated by the memory manager 1031 (Fig. 5) in response to requests from the instruction generator 1030 during instruction generation. It is the allocation of this memory for newly generated instructions which triggers the interaction between the memory manager 1031 and the queue manager 1032 described
10 below, and allows the system to adapt automatically to the amount of memory available and the complexity of the instructions involved.

The instruction queue manager 1032 is capable of waiting for the co-processor 224 to complete the execution of any given instruction which has been generated by the instruction generator 1030. However, by providing a
15 sufficiently large instruction queue 1022 and sufficient memory 203 for allocation by the memory manager 1031, it becomes possible to avoid having to wait for the co-processor 224 at all, or at least until the very end of the entire instruction sequence, which can be several minutes on a very large job. However, peak memory usage can easily exceed the memory available, and at
20 this point the interaction between the queue manager 1032 and the memory manager 1031 comes into play.

The instruction queue manager 1032 can be instructed at any time to "cleanup" the completed instructions by releasing the memory that was dynamically allocated for them. If the memory manager 1031 detects that
25 available memory is either running low or is exhausted, its first recourse is to instruct the queue manager 1032 to perform such a cleanup in an attempt to release some memory which is no longer in use by the co-processor 224. This can allow the memory manager 1031 to satisfy a request from the instruction generator 1030 for memory required by a newly generated instruction, without
30 the CPU 202 needing to wait for, or synchronize with, the co-processor 224.

If such a request made by the memory manager 1031 for the queue manager 1032 to cleanup completed instructions does not release adequate memory to satisfy the instruction generator's new request, the memory manager 1031 can request that the queue manager 1032 wait for a fraction, say half, of the outstanding instructions on the pending instruction queue 1040 to complete. This will cause the CPU 202 processing to block until some of the co-processor 224 instructions have been completed, at which point their operands can be freed, which can release sufficient memory to satisfy the request. Waiting for only a fraction of the outstanding instructions ensures that the co-processor 224 is kept busy by maintaining at least some instructions in its pending instruction queue 1040. In many cases the cleanup from the fraction of the pending instruction queue 1040 that the CPU 202 waits for, releases sufficient memory for the memory manager 1031 to satisfy the request from the instruction generator 1030.

In the unlikely event that waiting for the co-processor 224 to complete execution of, say, half of the pending instructions does not release sufficient memory to satisfy the request, then the final recourse of the memory manager 1031 is to wait until all pending co-processor instructions have completed. This should release sufficient resources to satisfy the request of the instruction generator 1030, except in the case of extremely large and complex jobs which exceed the system's present memory capacity altogether.

By the above described interaction between the memory manager 1031 and the queue manager 1032, the system effectively tunes itself to maximize throughput for the given amount of memory 203 available to the system. More memory results in less need for synchronization and hence greater throughput. Less memory requires the CPU 202 to wait more often for the co-processor 224 to finish using the scarce memory 203, thereby yielding a system which still functions with minimal memory available, but at a lower performance.

The steps taken by the memory manager 1031 when attempting to satisfy a request from the instruction generator 1030 are summarized below. Each step is tried in sequence, after which the memory manager 1031 checks to see if sufficient memory 203 has been made available to satisfy the request.

5 If so, it stops because the request can be satisfied; otherwise it proceeds to the next step in a more aggressive attempt to satisfy the request:

1. Attempt to satisfy the request with the memory 203 already available.
2. Cleanup all completed instructions.
- 10 3. Wait for a fraction of the pending instructions.
4. Wait for all the remaining pending instructions.

Other options can also be used in the attempt to satisfy the request, such as waiting for different fractions (such as one-third or two-thirds) of the pending instructions, or waiting for specific instructions which are known to be
15 using large amounts of memory.

Turning now to Fig. 7, in addition to the interaction between the memory manager 1031 and the queue manager 1032, the queue manager 1032 can also initiate a synchronization with the co-processor 224 in the case where space in a fixed-length instruction queue buffer 1050 is exhausted.
20 Such a situation is depicted in Fig. 7. In Fig. 7 the pending instructions queue 1040 is ten instructions in length. The latest instruction to be added to the queue 1040 has the highest occupied number. Thus where space is exhausted the latest instruction is located at position 9. The next instruction to be input to the co-processor 224 is waiting at position zero.

25 In such a case of exhausted space, the queue manager 1032 will also wait for, say, half the pending instructions to be completed by the co-processor 224. This delay normally allows sufficient space in the instruction queue 1040 to be freed for new instructions to be inserted by the queue manager 1032.

The method used by the queue manager 1032 when scheduling new instructions is as follows:

1. Test to see if sufficient space is available in the instruction queue 1040.
- 5 2. If sufficient space is not available, wait for the co-processor to complete some predetermined number or fraction of instructions.
3. Add the new instructions to the queue.

The method used by the queue manager 1032 when asked to wait for a given instruction is as follows:

- 10 1. Wait until the co-processor 224 indicates that the instruction is complete.
2. While there are instructions completed which are not yet cleaned up, clean up the next completed instruction in the queue.

The method used by the instruction generator 1030 when issuing new instructions is as follows:

1. Request sufficient memory for the instruction operands 1023 from the memory manager 1031.
2. Generate the instructions to be submitted.
3. Submit the co-processor instructions to the queue manager 1032 for execution.
- 20

The following is an example of pseudo code of the above decision making processes.

MEMORY MANAGER

25 *ALLOCATE_MEMORY*

BEGIN

IF sufficient memory is NOT available to satisfy request

THEN

Clean up all completed instructions.

30 *ENDIF*


```

      IF sufficient memory is still NOT available to satisfy
request
      THEN
          CALL WAIT_FOR_INSTRUCTION for half the
5 pending
          instructions.
      ENDIF
      IF sufficient memory is still NOT available to satisfy
request
10      THEN
          RETURN with an error.
      ENDIF
      RETURN the allocated memory
      END
15 QUEUE MANAGER
      SCHEDULE_INSTRUCTION
      BEGIN
          IF sufficient space is NOT available in the instruction queue
      THEN
20          WAIT for the co-processor to complete some
predetermined
          number of instructions.
      ENDIF
          Add the new instructions to the queue.
25      END
      WAIT_FOR_INSTRUCTION(i)
      BEGIN
          WAIT until the co-processor indicates that instruction i is
complete.

```

```

        WHILE there are instructions completed which are not yet
cleaned
        up
        DO
5           IF the next completed instruction has a cleanup
function
           THEN
               CALL the cleanup function
           ENDF
10          REMOVE the completed instruction from the queue
        DONE
    END
INSTRUCTION GENERATOR
    GENERATE_INSTRUCTIONS
15    BEGIN
        CALL ALLOCATE_MEMORY to allocate sufficient memory
for
        the instructions operands from the memory manager.
        GENERATE the instructions to be submitted.
20    CALL SCHEDULE_INSTRUCTION submit the co-
processor
        instructions to the queue manager for execution.
    END

```

25 3.3 Register Description of Co-processor

As explained above in relation to Figs. 1 and 3, the co-processor 224 maintains various registers 261 for the execution of each instruction stream.

Referring to each of the modules of Fig. 2, Table 1 sets out the name, type and description of each of the registers utilized by the co-processor 224
30 while Appendix B sets out the structure of each field of each register.

Table 1: Register Description

NAME	TYPE	DESCRIPTION
External Interface Controller Registers		
eic_cfg	Config2	Configuration
eic_stat	Status	Status
eic_err_int	Interrupt	Error and Interrupt Status
eic_err_int_en	Config2	Error and Interrupt Enable
eic_test	Config2	Test modes
eic_gen_pob	Config2	Generic bus programmable output bits
eic_high_addr	Config1	Dual address cycle offset
eic_wtlb_v	Control2	Virtual address and operation bits for TLB Invalidate/Write
eic_wtlb_p	Config2	Physical address and control bits for TLB Write
eic_mmu_v	Status	Most recent MMU virtual address translated, and current LRU location.
eic_mmu_v	Status	Most recent page table physical address fetched by MMU.
eic_ip_addr	Status	Physical address for most recent IBus access to the PCI Bus.
eic_rp_addr	Status	Physical address for most recent RBus access to the PCI Bus.
eic_ig_addr	Status	Address for most recent IBus access to the Generic Bus.
eic_rg_data	Status	Address for most recent RBus access to the Generic Bus.

NAME	TYPE	DESCRIPTION
Local Memory Controller Registers		
lmi_cfg	Control2	General configuration register
lmi_sts	Status	General status register
lmi_err_int	Interrupt	Error and interrupt status register
lmi_err_int_en	Control2	Error and interrupt enable register
lmi_dcfg	Control2	DRAM configuration register
lmi_mode	Control2	SDRAM mode register
Peripheral Interface Controller Registers		
pic_cfg	Config2	Configuration
pic_stat	Status	Status
pic_err_int	Interrupt	Interrupt/Error Status
pic_err_int_en	Config2	Interrupt/Error Enable
pic_abus_cfg	Control2	Configuration and control for ABUS
pic_abus_addr	Config1	Start address for ABUS transfer
pic_cent_cfg	Control2	Configuration and control for Centronics
pic_cent_dir	Config2	Centronics pin direct control register
pic_reverse_cfg	Control2	Configuration and control for reverse (input) data transfers
pic_timer0	Config1	Initial data timer value
pic_timer1	Config1	Subsequent data timer value
Miscellaneous Module Registers		
mm_cfg	Config2	Configuration Register
mm_stat	Status	Status Register
mm_err_int	Interrupt	Error and Interrupt Register
mm_err_int_en	Config2	Error and Interrupt Masks
mm_gefg	Config2	Global Configuration Register
mm_diag	Config	Diagnostic Configuration Register
mm_grst	Config	Global Reset Register
mm_gerr	Config2	Global Error Register
mm_gexp	Config2	Global Exception Register
mm_gint	Config2	Global Interrupt Register
mm_active	Status	Global Active signals

NAME	TYPE	DESCRIPTION
Instruction Controller Registers		
ic_cfg	Config2	Configuration Register
ic_stat	Status/ Interrupt	Status Register
ic_err_int	Interrupt	Error and Interrupt Register (write to clear error and interrupt)
ic_err_int_en	Config2	Error and Interrupt Enable Register
ic_ipa	Control1	A stream Instruction Pointer
ic_tda	Config1	A stream Todo Register
ic_fna	Control1	A stream Finished Register
ic_inta	Config1	A stream Interrupt Register
ic_loa	Status	A stream Last Overlapped Instruction Sequence number
ic_ipb	Control1	B stream Instruction Pointer
ic_tdb	Config1	B stream Todo Register
ic_fnb	Control1	B stream Finished Register
ic_intb	Config1	B stream Interrupt Register
ic_lob	Status	B stream Last Overlapped Instruction Sequence number
ic_sema	Status	A stream Semaphore
ic_semb	Status	B stream Semaphore
Data Cache Controller Registers		
dcc_cfg1	Config2	DCC configuration 1 register
dcc_stat	Status	state machine status bits
dcc_err_int	Status	DCC error status register
dcc_err_int_en	Control1	DCC error interrupt enable bits
dcc_cfg2	Control2	DCC configuration 2 register
dcc_addr	Config1	Base address register for special address modes.
dcc_lv0	Control1	"valid" bit status for lines 0 to 31
dcc_lv1	Control1	"valid" bit status for lines 32 to 63
dcc_lv2	Control1	"valid" bit status for lines 64 to 95
dcc_lv3	Control1	"valid" bit status for lines 96 to 127
dcc_raddrb	Status	Operand Organizer B request address
dcc_raddrc	Status	Operand Organizer C request address

NAME	TYPE	DESCRIPTION
dcc_test	Control1	DCC test register
Pixel Organizer Registers		
po_cfg	Config2	Configuration Register
po_stat	Status	Status Register
po_err_int	Interrupt	Error/Interrupt Status Register
po_err_int_en	Config2	Error/Interrupt Enable Register
po_dmr	Config2	Data Manipulation Register
po_subst	Config2	Substitution Value Register
po_cdp	Status	Current Data Pointer
po_len	Control1	Length Register
po_said	Control1	Start Address or Immediate Data
po_idr	Control2	Image Dimensions Register
po_muv_valid	Control2	MUV valid bits
po_muv	Config1	Base address of MUV RAM
Operand Organizer B Registers		
oob_cfg	Config2	Configuration Register
oob_stat	Status	Status Register
oob_err_int	Interrupt	Error/Interrupt Register
oob_err_int_en	Config2	Error/Interrupt Enable Register
oob_dmr	Config2	Data Manipulation Register
oob_subst	Config2	Substitution Value Register
oob_cdp	Status	Current Data Pointer
oob_len	Control1	Input Length Register
oob_said	Control1	Operand Start Address
oob_tile	Control1	Tiling length/offset Register
Operand Organizer C Registers		
ooc_cfg	Config2	Configuration Register
ooc_stat	Status	Status Register
ooc_err_int	Interrupt	Error/Interrupt Register
ooc_err_int_en	Config2	Error/Interrupt Enable Register
ooc_dmr	Config2	Data Manipulation Register
ooc_subst	Config2	Substitution Value Register
ooc_cdp	Status	Current Data Pointer
ooc_len	Control1	Input Length Register
ooc_said	Control1	Operand Start Address

NAME	TYPE	DESCRIPTION
ooc_tile	Control1	Tiling length/offset Register
JPEG Coder Register		
jc_cfg	Config2	configuration
jc_stat	Status	status
jc_err_int	Interrupt	error and interrupt status register
jc_err_int_en	Config2	error and interrupt enable register
jc_rsi	Config1	restart interval
jc_decode	Control2	decode of current instruction
jc_res	Control1	residual value
jc_table_sel	Control2	table selection from decoded instruction
Main Data Path Register		
mdp_cfg	Config2	configuration
mdp_stat	Status	status
mdp_err_int	Interrupt	error/interrupt
mdp_err_int_en	Config2	error/interrupt enable
mdp_test	Config2	test modes
mdp_op1	Control2	current operation 1
mdp_op2	Control2	current operation 2
mdp_por	Control1	offset for plus operator
mdp_bi	Control1	blend start/offset to index table entry
mdp_bm	Control1	blend end or number of rows and columns in matrix, binary places, and number of levels in halftoning
mdp_len	Control1	Length of blend to produce
Result Organizer Register		
ro_cfg	Config2	Configuration Register
ro_stat	Status	Status Register
ro_err_int	Interrupt	Error/Interrupt Register
ro_err_int_en	Config2	Error/Interrupt Enable Register
ro_dmr	Config2	Data Manipulation Register
ro_subst	Config1	Substitution Value Register
ro_cdp	Status	Current Data Pointer
ro_len	Status	Output Length Register
ro_sa	Config1	Start Address
ro_idr	Config1	Image Dimensions Register

NAME	TYPE	DESCRIPTION
ro_vbase	Config1	co-processor Virtual Base Address
ro_cut	Config1	Output Cut Register
ro_lmt	Config1	Output Length Limit
PCIBus Configuration Space alias		
		A read only copy of PCI configuration space registers 0x0 to 0xD and 0xF.
pci_external_cfg	Status	32-bit field downloaded at reset from an external serial ROM. Has no influence on coprocessor operation.
Input Interface Switch Registers		
iis_cfg	Config2	Configuration Register
iis_stat	Status	Status Register
iis_err_int	Interrupt	Interrupt/Error Status Register
iis_err_int_en	Config2	Interrupt/Error Enable Register
iis_ic_addr	Status	Input address from IC
iis_dcc_addr	Status	Input address from DCC
iis_po_addr	Status	Input address from PO
iis_burst	Status	Burst Length from PO, DCC & IC
iis_base_addr	Config1	Base address of co-processor memory object in host memory map.
iis_test	Config1	Test mode register

The more notable ones of these registers include:

- (a) Instruction Pointer Registers (ic_ipa and ic_ipb). This pair of registers each contains the virtual address of the currently executing instruction. Instructions are fetched from ascending virtual addresses and executed. Jump instruction can be used to transfer control across non-contiguous virtual addresses. Associated with each instruction is a 32 bit sequence number which increments by one per instruction. The sequence numbers are used by both the co-processor 224 and by the host CPU 202 to synchronize instruction generation and execution.
- (b) Finished Registers (ic_fna and ic_fnb). This pair of registers each contains a sequence number counting completed instructions.
- (c) Todo Register (ic_tda and ic_tdb). This pair of registers each contains a sequence number counting queued instructions.

(d) Interrupt Register (ic_inta and ic_intb). This pair of registers each contains a sequence number at which to interrupt.

(e) Interrupt Status Registers (ic_stat.a_primed and ic_stat.b_primed). This pair of registers each contains a primed bit which is
 5 a flag enabling the interrupt following a match of the Interrupt and Finished Registers. This bit appears alongside other interrupt enable bits and other status/configuration information in the Interrupt Status (ic_stat) register.

(f) Register Access Semaphores (ic_sema and ic_semb). The host CPU 202 must obtain this semaphore before attempting register
 10 accesses to the co-processor 224 that requires atomicity, ie. more than one register write. Any register accesses *not* requiring atomicity can be performed at any time. A side effect of the host CPU 202 obtaining this semaphore is that co-processor execution pauses once the currently executing instruction has completed. The Register Access Semaphore is implemented
 15 as one bit of the configuration/status register of the co-processor 224. These registers are stored in the Instruction Controllers own register area. As noted previously, each sub-module of the co-processor has its own set of configuration and status registers. These registers are set in the course of regular instruction execution. All of these registers appear in the register
 20 map and many are modified implicitly as part of instruction execution. These are all visible to the host via the register map.

3.4 Format of Plural Streams

As noted previously, the co-processor 224, in order to maximize the utilization of its resources and to provide for rapid output on any external
 25 peripheral device, executes one of two independent instruction streams. Typically, one instruction stream is associated with a current output page required by an output device in a timely manner, while the second instruction stream utilizes the modules of the co-processor 224 when the other instruction stream is dormant. Clearly, the overriding imperatives are to provide the
 30 required output data in a timely manner whilst simultaneously attempting to

maximize the use of resources for the preparation of subsequent pages, bands, etc. The co-processor 224 is therefore designed to execute two completely independent but identically implemented instruction streams (hereafter termed A and B). The instructions are preferably generated by software
 5 running on the host CPU 202 (Fig. 1) and forwarded to the raster image acceleration card 220 for execution by the co-processor 224. One of the instruction streams (stream A) operates at a higher priority than the other instruction stream (stream B) during normal operation. The stream or queue of instructions is written into a buffer or list of buffers within the host
 10 RAM 203 (Fig. 1) by the host CPU 202. The buffers are allocated at start-up time and locked into the physical memory of the host 203 for the duration of the application. Each instruction is preferably stored in the virtual memory environment of the host RAM 203 and the raster image co-processor 224 utilizes a virtual to physical address translation scheme to determine a
 15 corresponding physical address with the in-host RAM 203 for the location of a next instruction. These instructions may alternatively be stored in the co-processors 224 local memory.

Turning now to Fig. 8, there is illustrated the format of two instruction streams A and B 270, 271 which are stored within the host RAM 203. The
 20 format of each of the streams A and B is substantially identical.

Briefly, the execution model for the co-processor 224 consists of:

- Two virtual streams of instructions, the A stream and the B stream.
- In general only one instruction is executed at a time.
- 25 • Either stream can have priority, or priority can be by way of "round robin".
- Either stream can be "locked" in, ie. guaranteed to be executed regardless of stream priorities or availability of instructions on the other stream.

- Either stream can be empty.
- Either stream can be disabled.
- Either stream can contain instructions that can be "overlapped", ie. execution of the instruction can be overlapped with that of the following instruction if the following instruction is not also "overlapped".
- Each instruction has a "unique" 32 bit incrementing sequence number.
- Each instruction can be coded to cause an interrupt, and/or a pause in instruction execution.
- Instructions can be speculatively prefetched to minimize the impact of external interface latency.

The instruction controller 235 is responsible for implementing the co-processor's instruction execution model maintaining overall executive control of the co-processor 224 and fetching instructions from the host RAM 203 when required. On a per instruction basis, the instruction controller 235 carries out the instruction decoding and configures the various registers within the modules via CBus 231 to force the corresponding modules to carry-out that instruction.

Turning now to Fig. 9, there is illustrated a simplified form of the instruction execution cycle carried out by the instructions controller 235. The instruction execution cycle consists of four main stages 276-279. The first stage 276 is to determine if an instruction is pending on any instruction stream. If this is the case, an instruction is fetched 277, decoded and executed 278 by means of updating registers 279.

3.5 Determine Current Active Stream

In implementing the first stage 276, there are two steps which must be taken:

1. Determine whether an instruction is pending; and
2. Decide which stream of instructions should be fetched next.

In determining whether instructions are pending the following possible conditions must be examined:

1. whether the instruction controller is enabled;
2. whether the instruction controller is paused due to an internal
5 error or interrupt;
3. whether there is any external error condition pending;
4. whether either of the A or B streams are locked;
5. whether either stream sequence numbering is enabled; and
6. whether either stream contains a pending instruction.

10 The following pseudo code describes the algorithm for determining whether an instruction is pending in accordance with the above rules. This algorithm can be hardware implemented via a state transition machine within the instruction controller 235 in known manner:

```

15      if not error and enabled and not bypassed and not self test mode
           if A stream locked and not paused
                if A stream enabled and (A stream
sequencing disabled or instruction on A stream)
                     instruction pending
20      else
                     no instruction pending
                     end if
           else
           if B stream locked and not paused
25      if B stream enabled and (B stream
sequencing disabled or instruction on B stream)
                     instruction pending
                     else
                     no instruction pending
30      end if

```

```

                                clsc                                /* no stream is locked */
                                if (A stream enabled and not paused
                                and (A stream sequencing disabled or instruction on A stream))
                                or (B stream enabled and not paused
5    and (B stream sequencing disabled or instruction on B stream))
                                instruction pending
                                else
                                no instruction pending
                                end if
10    end if
                                else                                /* interface controller not enabled */
                                no instruction pending
                                end if

```

15 If no instruction is found pending, then the instruction controller 235 will "spin" or idle until a pending instruction is found.

 To determine which stream is "active", and which stream is executed next, the following possible conditions are examined:

1. whether either stream is locked;
- 20 2. what priority is given to the A and B streams and what the last instruction stream was;
3. whether either stream is enabled; and
4. whether either stream contains a pending instruction.

 The following pseudo code implemented by the instruction controller
25 describes how to determine the next active instruction stream:

```

                                if A stream locked
                                next stream is A
                                else if B stream locked
30    next stream is B

```

```

else                                     /* no stream is locked */
    if (A stream enabled and (A stream sequencing disabled or
instruction on A stream)) and not (B stream enabled and (B stream
sequencing disabled or instruction on B stream))
5        next stream is A
    else if (B stream enabled and (B stream sequencing disabled
or instruction on B stream)) and not (A stream enabled and (A stream
sequencing disabled or instruction on A stream))
        next stream is B
10    else                               /* both stream have instruction */
        if pri = 0    /* A high, B low */
            next stream is A
        else if pri = 1 /* A low, B high */
            next stream is B
15    else if pri = 2 or 3 /* round robin */
        if last stream is A
            next stream is B
        else
            next stream is A
20    end if
    end if
end if
end if

```

25 As the conditions can be constantly changing, all conditions must be determined together atomically.

3.6 Fetch Instruction of Current Active Stream

After the next active instruction stream is determined, the Instruction Controller 235 fetches the instruction using the address in the corresponding
30 instruction pointer register (ic_ipa or ic_ipb). However, the Instruction

Controller 235 does not fetch an instruction if a valid instruction already exists in a prefetch buffer stored within the instruction controller 235.

A valid instruction is in the prefetch buffer if:

1. the prefetch buffer is valid; and
- 5 2. the instruction in the prefetch buffer is from the same stream as the currently active stream.

The validity of the contents of the prefetch buffer is indicated by a prefetch bit in the ic_stat register, which is set on a successful instruction prefetch. Any external write to any of the registers of the instruction
10 controller 235 causes the contents of the prefetch buffer to be invalidated.

3.7 Decode and Execute Instruction

Once an instruction has been fetched and accepted the instruction controller 235 decodes it and configures the registers 229 of the co-processor 224 to execute the instruction.

15 The instruction format utilized by the raster image co-processor 224 differs from traditional processor instruction sets in that the instruction generation must be carried out instruction by instruction by the host CPU 202 and as such is a direct overhead for the host. Further, the instructions should be as small as possible as they must be stored in host RAM 203 and
20 transferred over the PCI bus 206 of Fig. 1 to the co-processor 224. Preferably, the co-processor 224 can be set up for operation with only one instruction. As much flexibility as possible should be maintained by the instruction set to maximize the scope of any future changes. Further, preferably any instruction executed by the co-processor 224 applies to a long stream of
25 operand data to thereby achieve best performance. The co-processor 224 employs an instruction decoding philosophy designed to facilitate simple and fast decoding for "typical instructions" yet still enable the host system to apply a finer control over the operation of the co-processor 224 for "atypical" operations.

Turning now to Fig. 10, there is illustrated the format of a single instruction 280 which comprizes eight words each of 32 bits. Each instruction includes an instruction word or opcode 281, and an operand or result type data word 282 setting out the format of the operands. The
 5 addresses 283-285 of three operands A, B and C are also provided, in addition to a result address 286. Further, an area 287 is provided for use by the host CPU 202 for storing information relevant to the instruction.

The structure 290 of an instruction opcode 281 of an instruction is illustrated in Fig. 11. The instruction opcode is 32 bits long and includes a
 10 major opcode 291, a minor opcode 292, an interrupt (I) bit 293, a partial decode (Pd) bit 294, a register length (R) bit 295, a lock (L) bit 296 and a length 297. A description of the fields in the instruction word 290 is as provided by the following table.

15

Table 2: Opcode Description

Field	Description
major opcode [3..0]	Instruction category 0: Reserved 1: General Colour Space Conversion 2: JPEG Compression and Decompression 3: Matrix Multiplication 4: Image Convolutions 5: Image Transformations 6: Data Coding 7: Halftone 8: Hierarchical image decompression 9: Memory Copy 10: Internal Register and Memory Access 11: Instruction Flow Control 12: Compositing 13: Compositing 14: Reserved 15: Reserved
minor opcode [7..0]	Instruction detail. The coding of this field is dependent on the major opcode.
I	1 = Interrupt and pause when completed, 0 = Don't interrupt and pause when completed

Field	Description
pd	Partial Decode 1 = use the "partial decode" mechanism. 0 = Don't use the "partial decode" mechanism
R	1 = length of instruction is specified by the Pixel Organizer's input length register (po_len) 0 = length of instruction is specified by the opcode length field.
L	1 = this instruction stream (A or B) is "locked" for the next instruction. 0 = this instruction stream (A or B) is not "locked" in for the next instruction.
length [15..0]	number of data items to read or generate

By way of discussion of the various fields of an opcode, by setting the I.
 5 bit field 293, the instruction can be coded such that instruction execution sets

an interrupt and pause on completion of that instruction. This interrupt is called an "instruction completed interrupt". The partial decode bit 294 provides for a partial decode mechanism such that when the bit is set and also enabled in the ic_cfg register, the various modules can be micro coded
 5 prior to the execution of the instruction in a manner which will be explained in more detail hereinafter. The lock bit 296 can be utilized for operations which require more than one instruction to set up. This can involve setting various registers prior to an instruction and provides the ability to "lock" in the current instruction stream for the next instruction. When the L-bit 296 is set,
 10 once an instruction is completed, the next instruction is fetched from the same stream. The length field 297 has a natural definition for each instruction and is defined in terms of the number of "input data items" or the number of "output data items" as required. The length field 297 is only 16 bits long. For instructions operating on a stream of input data items greater than
 15 64,000 items the R-bit 295 can be set, in which case the input length is taken from a po_len register within the pixel organizer 246 of Fig. 2. This register is set immediately before such an instruction.

Returning to Fig. 10, the number of operands 283-286 required for a given instruction varies somewhat depending on the type of instruction
 20 utilized. The following table sets out the number of operands and length definition for each instruction type:

Table 3: Operand Types

Instruction Class	Length defined by		# of operand s
Compositing	input	pixels	3
General Color Space Conversion	input	pixels	2
JPEG decompression/compression	input	bytes	2
other decompression/compression	input	bytes	2
Image Transformations and Convolutions	output	bytes	2
Matrix Multiplication	input	pixels	2
Halftoning	input	pixels, bytes	2
Memory Copying	input	pixels, bytes	1
Hierarchical Image Decompression	input	pixels, bytes	1 or 2
Flow Control	fixed	fixed	2
Internal Access Instructions	fixed	fixed	4

Turning now to Fig. 12, there is illustrated, firstly, the data word format 300 of the data word or operand descriptor 282 of Fig. 10 for three operand instructions and, secondly, the data word format 301 for two operand instructions. The details of the encoding of the operand descriptors are provided in the following table:

Table 4: Operand Descriptors

Field	Description
what	<p>0 = instruction specific mode: This indicates that the remaining fields of the descriptor will be interpreted in line with the major opcode. Instruction specific modes supported are: major opcode = 0-11: Reserved major opcode = 12-13: (Compositing): Implies that Operand C is a bitmap attenuation. The occ_dmr register will be set appropriately, with the cc=1 and normalize=0 major opcode = 14-15: Reserved 1 = sequential addressing 2 = tile addressing 3 = constant data</p>

Field	Description
L	0 = not long: immediate data 1 = long: pointer to data
if	internal format: 0 = pixels 1 = unpacked bytes 2 = packed bytes 3 = other
S	0 = set up Data Manipulation Register as appropriate for this operand 1 = use the Data Manipulation Register as is
C	0 = not cacheable 1 = cacheable Note: In general a performance gain will be achieved if an operand is specified as cacheable. Even operands displaying low levels of referencing locality (such as sequential data) still benefit from being cached - as it allows data to be burst transferred to the host processor and is more efficient.
P	external format: 0 = unpacked bytes 1 = packed stream
bo[2:0]	bit offset. Specifies the offset within a byte of the start of bitwise data.
R	0 = Operand C does not describe a register to set. 1 = Operand C describes a register to set. This bit is only relevant for instructions with less than three operands.

With reference to the above table, it should be noted that, firstly, in respect of the constant data addressing mode, the co-processor 224 is set up to
 5 fetch, or otherwise calculate, one internal data item, and use this item for the length of the instruction for that operand. In the tile addressing mode, the co-processor 224 is set up to cycle through a small set of data producing a "tiling effect". When the L-bit of an operand descriptor is zero then the data is immediate, ie. the data items appear literally in the operand word.

Returning again to Fig. 10, each of the operand and result words 283-286 contains either the value of the operand itself or a 32-bit virtual address to the start of the operand or result where data is to be found or stored.

The instruction controller 235 of Fig. 2 proceeds to decode the instruction in two stages. It first checks to see whether the major opcode of the instruction is valid, raising an error if the major opcode 291 (Fig. 11) is invalid. Next, the instruction is executed by the instruction controller 235 by means of setting the various registers via CBus 231 to reflect the operation specified by the instruction. Some instructions can require no registers to be set.

The registers for each module can be classified into types based on their behavior. Firstly, there is the status register type which is "read only" by other modules and "read/write" by the module including the register. Next, a first type of configuration register, hereinafter called "config1", is "read/write" externally by the modules and "read only" by the module including the register. These registers are normally used for holding larger type configuration information, such as address values. A second type of configuration register, herein known as "config2", is readable and writable by any module but is read only by the module including the register. This type of register is utilized where bit by bit addressing of the register is required.

A number of control type registers are provided. A first type, hereinafter known as "control1" registers, is readable and writable by all modules (including the module which includes the register). The control1 registers are utilized for holding large control information such as address values. Analogously, there is further provided a second type of control register, hereinafter known as "control2", which can be set on a bit by bit basis.

A final type of register known as an interrupt register has bits within the register which are settable to 1 by the module including the register and resettable to zero externally by writing a "1" to the bit that has been set.

This type of register is utilized for dealing with the interrupts/errors flagged by each of the modules.

Each of the modules of the co-processor 224 sets a `c_active` line on the CBus 231 when it is busy executing an instruction. The instruction controller 235 can then determine when instructions have been completed by "OR-ing" the `c_active` lines coming from each of the modules over the CBus 231. The local memory controller module 236 and the peripheral interface controller module 237 are able to execute overlapped instructions and include a `c_background` line which is activated when they are executing an overlapped instruction. The overlapped instructions are "local DMA" instructions transferring data between the local memory interface and the peripheral interface.

The execution cycle for an overlapped local DMA instruction is slightly different from the execution cycle of other instructions. If an overlapped instruction is encountered for execution, the instruction controller 235 checks whether there is already an overlapped instruction executing. If there is, or overlapping is disabled, the instruction controller 235 waits for that instruction to finish before proceeding with execution of that instruction. If there is not, and overlapping is enabled, the instruction controller 235 immediately decodes the overlapped instruction and configures the peripheral interface controller 237 and local memory controller 236 to carry out the instruction. After the register configuration is completed, the instruction controller 235 then goes on to update its registers (including finished register, status register, instruction pointer, etc.) without waiting for the instruction to "complete" in the conventional sense. At this moment, if the finished sequence number equals the interrupt sequence number, the overlapped instruction completed' interrupt is primed rather than raising the interrupt immediately. The 'overlapped instruction completed' interrupt is raised when the overlapped instruction has fully completed.

Once the instruction has been decoded, the instruction controller attempts to prefetch the next instruction while the current instruction is executing. Most instructions take considerably longer to execute than they will to fetch and decode. The instruction controller 235 prefetches an
 5 instruction if all of the following conditions are met:

1. the currently executing instruction is not set to interrupt and pause;
2. the currently executing instruction is not a jump instruction;
3. the next instruction stream is prefetch-enabled; and
- 10 4. there is another instruction pending.

If the instruction controller 235 determines that prefetching is possible it requests the next instruction, places it in a prefetch buffer and then validates the buffer. At this point there is nothing more for the instruction controller 235 to do until the currently executing instruction has
 15 completed. The instruction controller 235 determines the completion of an instruction by examining the c_active and c_background lines associated with the CBus 231.

3.8 Update Registers of Instruction Controller

Upon completion of an instruction, the instruction controller 235
 20 updates its registers to reflect the new state. This must be done atomically to avoid problems with synchronising with possible external accesses. This atomic update process involves:

1. Obtaining the appropriate Register Access Semaphore. If the semaphore is taken by an agent external to the Instruction Controller 235, the
 25 instruction execution cycle waits at this point for the semaphore to be released before proceeding.

2. Updating the appropriate registers. The instruction pointer (ic_ipa or ic_ipb) is incremented by the size of an instruction, unless the instruction was a successful jump, in which case the target value of the jump
 30 is loaded into the instruction pointer.

The finished register (ic_fna or ic_fnb), is then incremented if sequence numbering is enabled.

The status register (ic_stat) is also updated appropriately to reflect the new state. This includes setting the pause bits if necessary. The
 5 Instruction Controller 235 pauses if an interrupt has occurred and pausing is enabled for that interrupt or if any error has occurred. Pausing is implemented by setting the instruction stream pause bits in the status register (a_pause or b_pause bits in ic_stat). To resume instruction execution, these bits should be reset to 0.

10 3. Asserting a c_end signal on the CBus 231 for one clock cycle, which indicates to other modules in the co-processor 224 that an instruction has been completed.

4. Raising an interrupt if required. An interrupt is raised if:

a. "Sequence number completed" interrupt occurs. That is, if
 15 the finished register (ic_fna or ic_fnb) sequence number is the same as interrupt sequence number. Then this interrupt is primed, sequence numbering is enabled, and the interrupt occurs; or

b. the just completed instruction was coded to interrupt on completion, then this mechanism is enabled.

20 3.9 Semantics of the Register Access Semaphore

The Register Access Semaphore is a mechanism that provides atomic accesses to multiple instruction controller registers. The registers that can require atomic access are as follows:

1. Instruction pointer register (ic_ipa and ic_ipb)
- 25 2. Todo registers (ic_tda and ic_tdb)
3. Finished registers (ic_fna and ic_fnb)
4. Interrupt registers (ic_inta and ic_intb)
5. The pause bits in the configuration register (ic_cfg)

External agents can read all registers safely at any time. External
 30 agents are able to write any registers at any time, however to ensure that the

Instruction Controller 235 does not update values in these registers, the external agent must first obtain the Register Access Semaphore. The Instruction Controller does not attempt to update any values in the abovementioned registers if the Register Access Semaphore is claimed externally. The instruction controller 235 updates all of the above mentioned registers in one clock cycle to ensure atomicity.

As mentioned above, unless the mechanism is disabled, each instruction has associated with it a 32 bit "sequence number". Instruction sequence numbers increment wrapping through from 0xFFFFFFFF to 0x00000000.

When an external write is made into one of the Interrupt Registers (ic_inta or ic_intb), the instruction controller 235 immediately makes the following comparisons and updates:

1. If the interrupt sequence number (ie. the value in the Interrupt Register) is "greater" (in a modulo sense) than the finished sequence number (ie. the value in the Finished Register) of the same stream, the instruction controller primes the "sequence number completed" interrupt mechanism by setting the "sequence number completed" primed bit (a_primed or b_primed bit in ic_stat) in the status register.
2. If the interrupt sequence number is not "greater" than the finished sequence number, but there is an overlapped instruction in progress in that stream and the interrupt sequence number equals the last overlapped instruction sequence number (ie. the value in the ic_loa or ic_lob register), then the instruction controller primes the "overlapped instruction sequence number completed" interrupt mechanism by setting the a_ol_primed or b_ol_primed bits in the ic_stat register.
3. If the interrupt sequence number is not "greater" than the finished sequence number, and there is an overlapped instruction in progress in that stream, but the interrupt sequence number does not equal the last

overlapped instruction sequence number, then the interrupt sequence number represents a finished instruction, and no interrupt mechanism is primed.

4. If the interrupt sequence number is not "greater" than the finished sequence number, and there is no overlapped instruction in progress in that stream, then the interrupt sequence number must represent a finished instruction, and no interrupt mechanism is primed.

External agents can set any of the interrupt primed bits (bits `a_primed`, `a_ol_primed`, `b_primed` or `b_ol_primed`) in the status register to activate or de-activate this interrupt mechanism independently.

10 3.10 Instruction Controller

Turning now to Fig. 13, there is illustrated the instruction controller 235 in more detail. The instruction controller 235 includes an execution controller 305 which implements the instruction execution cycle as well as maintaining overall executive control of the co-processor 224. The functions of the execution controller 305 include maintaining overall executive control of the instruction controller 235, determining instructing sequencing, instigating instruction fetching and prefetching, initiating instructing decoding and updating the instruction controller registers. The instruction controller further includes an instruction decoder 306. The instruction decoder 306 accepts instructions from a prefetch buffer controller 307 and decodes them according the aforementioned description. The instruction decoder 306 is responsible for configuring registers in the other co-processor modules to execute the instruction. The prefetch buffer controller 307 manages the reading and writing to a prefetch buffer within the prefetch buffer controller and manages the interfacing between the instruction decoder 306 and the input interface switch 252 (Fig. 2). The prefetch buffer controller 307 is also responsible for managing the updating of the two instruction pointer registers (`ic_ipa` and `ic_ipb`). Access to the CBus 231 (Fig. 2) by the instruction controller 235, the miscellaneous module 239 (Fig. 2) and the external interface controller 238 (Fig. 2) is controlled by a "CBus" arbitrator 308 which

arbitrates between the three modules' request for access. The requests are transferred by means of a control bus (CBus) 231 to the register units of the various modules.

Turning now to Fig. 14, there is illustrated the execution controller 305 of Fig. 13 in more detail. As noted previously, the execution controller is responsible for implementing the instruction execution cycle 275 of Fig. 9 and, in particular, is responsible for:

1. Determining which instruction stream the next instruction is to come from;
- 10 2. Initiating fetching of that instruction;
3. Signalling the instruction decoder to decode the instruction as residing in the prefetch buffer;
4. Determining and initiating any prefetching of the next instruction;
- 15 5. Determining instruction completion; and
6. Updating the registers after the instruction has completed.

The execution controller includes a large core state machine 310 hereinafter known as "the central brain" which implements the overall instruction execution cycle. Turning to Fig. 15, there is illustrated the state machine diagram for the central brain 310 implementing the instruction execution cycle as aforementioned. Returning to Fig. 14, the execution controller includes an instruction prefetch logic unit 311. This unit is responsible for determining whether there is an outstanding instruction to be executed and which instruction stream the instruction belongs to. The start 312 and prefetch 313 states of the transition diagram of Fig. 15 utilize this information in obtaining instructions. A register management unit 317 of Fig. 14 is responsible for monitoring the register access semaphores on both instruction streams and updating all necessary registers in each module. The register management unit 317 is also responsible for comparing the finished register (ic_fna or ic_fnb) with the interrupt register (ic_inta or

ic_intb) to determine if a "sequence number completed" interrupt is due. The register management unit 317 is also responsible for interrupt priming. An overlapped instructions unit 318 is responsible for managing the finishing off of an overlapped instruction through management of the appropriate status bits in the ic_stat register. The execution controller also includes a decoder interface unit 319 for interfacing between the central brain 310 and the instruction decoder 306 of Fig. 13.

Turning now to Fig. 16, there is illustrated the instruction decoder 306 in more detail. The instruction decoder is responsible for configuring the co-processor to execute the instructions residing in the prefetch buffer. The instruction decoder 306 includes an instruction decoder sequencer 321 which comprizes one large state machines broken down into many smaller state machines. The instruction sequencer 321 communicates with a CBus dispatcher 312 which is responsible for setting the registers within each module. The instruction decoder sequencer 321 also communicates relevant information to the execution controller such as instruction validity and instruction overlap conditions. The instruction validity check being to check that the instruction opcode is not one of the reserved opcodes.

Turning now to Fig. 17, there is illustrated, in more detail, the instruction dispatch sequencer 321 of Fig. 16. The instruction dispatch sequencer 321 includes a overall sequencing control state machine 324 and a series of per module configuration sequencer state machines, eg. 325, 326. One per module configuration sequencer state machine is provided for each module to be configured. Collectively the state machines implement the co-processor's microprogramming of the modules. The state machines, eg. 325, instruct the CBus dispatcher to utilize the global CBus to set various registers so as to configure the various modules for processing. A side effect of writing to particular registers is that the instruction execution commences. Instruction execution typically takes much longer than the time it takes for the sequencer 321 to configure the co-processor registers for execution. In

appendix A, attached to the present specification, there is disclosed the microprogramming operations performed by the instruction sequencer of the co-processor in addition to the form of set up by the instruction sequencer 321.

In practice, the Instruction Decode Sequencer 321 does not configure
 5 all of the modules within the co-processor for every instruction. The table below shows the ordering of module configuration for each class of instruction with the module configured including the pixel organizer 246 (PO), the data cache controller 240 (DCC), the operand organizer B 247 (OOB), the operand organizer C 248 (OOC), main data path 242 (MDP), results organizer 249
 10 (RO), and JPEG encoder 241 (JC). Some of the modules are never configured during the course of instruction decoding. These modules are the External Interface Controller 238 (EIC), the Local Memory Controller 236 (LMC), the Instruction Controller 235 itself (IC), the Input Interface Switch 252 (IIS) and the Miscellaneous Module (MM).

15

Table 5: Module Setup Order

Instruction Class	Module Configuration Sequence	Sequence ID
Compositing	PO, DCC, OOB, OOC, MDP, RO	1
CSC	PO, DCC, OOB, OOC, MDP, RO	2
JPEG coding	PO, DCC, OOB, OOC, JC, RO	3
Data coding	PO, DCC, OOB, OOC, JC, RO	3
Transformations and Convolutions	PO, DCC, OOB, OOC, MDP, RO	2
Matrix Multiplication	PO, DCC, OOB, OOC, MDP, RO	2
Halftoning	PO, DCC, OOB, MDP, RO	4
General memory copy	PO, JC, RO	8
Peripheral DMA	PIC	5
Hierarchical Image - Horizontal Interpolation	PO, DCC, OOB, OOC, MDP, RO	6
Hierarchical Image - others	PO, DCC, OOB, OOC, MDP, RO	4
Internal access	RO, RO, RO, RO	7
others	-	-

Turning now to Fig. 17, each of the module configuration sequencers, eg. 325 is responsible for carrying out the required register access operations to configure the particular module. The overall sequencing control state machine 324 is responsible for overall operation of the module configuration sequencer in the aforementioned order.

Referring now to Fig. 18, there is illustrated 330 the state transition diagram for the overall sequencing control unit which basically activates the relevant module configuration sequencer in accordance with the above table. Each of the modules configuration sequencers is responsible for controlling the CBus dispatcher to alter register details in order to set the various registers in operation of the modules.

Turning now to Fig. 19, there is illustrated the prefetch buffer controller 307 of Fig. 13 in more detail. The prefetch buffer controller consists of a prefetch buffer 335 for the storage of a single co-processor instruction (six times 32 bit words). The prefetch buffer includes one write
 5 port controlled by a IBus sequencer 336 and one read port which provides data to the instruction decoder, execution controller and the instruction controller CBus interface. The IBus sequencer 336 is responsible for observing bus protocols in the connection of the prefetch buffer 335 to the input interface switch. An address manager unit 337 is also provided which deals with
 10 address generation for instruction fetching. The address manager unit 337 performs the functions of selecting one of ic_ipa or ic_ipb to place on the bus to the input interface switch, incrementing one of ic_ipa or ic_ipb based on which stream the last instructions was fetched from and channelling jump target addresses back to the ic_ipa and ic_ipb register. A PBC controller 339
 15 maintains overall control of the prefetched buffer controller 307.

3.11 Description of a Modules Local Register File

As illustrated in Fig. 13, each module, including the instruction controller module itself, has an internal set of registers 304 as previously defined in addition to a CBus interface controller 303 as illustrated in Fig. 20
 20 and which is responsible for receiving CBus requests and updating internal registers in light of those requests. The module is controlled by writing registers 304 within the module via a CBus interface 302. A CBus arbitrator 308 (Fig. 13) is responsible for determining which module of the instruction controller 235, the external interface controller or the
 25 miscellaneous module is able to control the CBus 309 for acting as a master of the CBus and for the writing or reading of registers.

Fig. 20, illustrates, in more detail, the standard structure of a CBus interface 303 as utilized by each of the modules. The standard CBus interface 303 accepts read and write requests from the CBus 302 and includes
 30 a register file 304 which is utilized 341 and updated on 341 by the various

submodules within a module. Further, control lines 344 are provided for the updating of any submodule memory areas including reading of the memory areas. The standard CBus interface 303 acts as a destination on the CBus, accepting read and write requests for the register 304 and memory objects
 5 inside other submodules.

A "c_reset" signal 345 sets every register inside the Standard CBus interface 103 to their default states. However, "c_reset" will not reset the state machine that controls the handshaking of signals between itself and the CBus Master, so even if "c_reset" is asserted in the middle of a CBus
 10 transaction, the transaction will still finish, with undefined effects. The "c_int" 347, "c_exp" 348 and "c_err" 349 signals are generated from the content of a modules err_int and err_int_en registers by the following equations:

$$15 \quad c_err = \sum_{\text{error}[i] \text{ not reserved}} \text{error}[i] \text{ AND err_mask}[i] \quad (1)$$

$$c_int = \sum_{\text{interrupt}[i] \text{ not reserved}} \text{interrupt}[i] \text{ AND int_mask}[i] \quad (2)$$

$$20 \quad c_exp = \sum_{[i] \text{ not reserved}} \text{exception}[i] \text{ AND exp_mask}[i] \quad (3)$$

The signals "c_sdata_in" 345 and "c_svalid_in" are data and valid signals from the previous module in a daisy chain of modules. The signals "c_sdata_out" and "c_svalid_out"
 25 350 are data and valid signals going to the next module in the daisy chain.

The functionality of the Standard CBus interface 303 includes:

1. register read/write handling
2. memory area read/write handling
3. test mode read/write handling
- 30 4. submodule observe/update handling

3.12 Register Read/Write Handling

The Standard CBus Interface 303 accepts register read/write and bit set requests that appears on the CBus. There are two types of CBus instructions that Standard CBus Interface handles:

5 1. Type A

Type A operations allow other modules to read or write 1, 2, 3, or 4 bytes into any register inside Standard CBus Interface 303. For write operations, the data cycle occurs in the clock cycle immediately after the instruction cycle. Note that the type field for register write and read are
 10 "1000" and "1001" respectively. The Standard CBus Interface 303 decodes the instruction to check whether the instruction is addressed to the module, and whether it is a read or write operation. For read operation, the Standard CBus Interface 303 uses the "reg" field of the CBus transaction to select which register output is to put into the "c_sdata" bus 350. For write operations, the
 15 Standard CBus Interface 303 uses the "reg" and "byte" fields to write the data into the selected register. After read operation is completed, the Standard CBus Interface returns the data and asserts "c_svalid" 350 at the same time. After write operations are completed, the Standard CBus Interface 303 asserts "c_svalid" 350 to acknowledge.

20 2. Type C

Type C operations allow other modules to write one or more bits in one of the bytes in one of the registers. Instruction and data are packed into one word.

The Standard CBus Interface 303 decodes the instruction to check
 25 whether the instruction is addressed to the module. It also decodes "reg", "byte" and "enable" fields to generate the required enable signals. It also latches the data field of the instruction, and distributes it to all four bytes of a word so the required bit(s) are written in every enabled bit(s) in every enabled byte(s). No acknowledgment is required for this operation.

3.13 Memory Area Read/Write Handling

The Standard CBus Interface 303 accepts memory read and memory write requests that appears on the CBus. While accepting a memory read/write request, the Standard CBus Interface 303 checks whether the request is addressed to the module. Then, by decoding the address field in the instruction, the Standard CBus Interface generates the appropriate address and address strobe signals 344 to the submodule which a memory read/write operation is addressed to. For write operations the Standard CBus Interface also passes on the byte enable signals from the instruction to the submodules.

The operation of the standard CBus interface 303 is controlled by a read/write controller 352 which decodes the type field of a CBus instruction from the CBus 302 and generates the appropriate enable signals to the register file 304 and output selector 353 so that the data is latched on the next cycle into the register file 304 or forwarded to other submodules 344. If the CBus instruction is a register read operation, the read/write controller 352 enables the output selector 353 to select the correct register output going onto the "c_sdata bus" 345. If the instruction is a register write operation, the read/write controller 352 enables the register file 304 to select the data in the next cycle. If the instruction is a memory area read or write, then the read/write controller 352 generates the appropriate signals 344 to control those memory areas under a modules control. The register file 304 contains four parts, being a register select decoder 355, an output selector 353, interrupt 356, error 357 and exception 358 generators, unmasked error generator 359 and the register components 360 which make up the registers of that particular module. The register select decoder 355 decodes the signal "ref_en" (register file enable), "write" and "reg" from the read/write controller 352 and generates the register enable signals for enabling the particular register of interest. The output selector 353 selects the correct register data

to be output on c_sdata_out lines 350 for register read operations according to the signal "reg" output from the read/write controller 352.

The exception generators 356-359 generate an output error signal, eg. 347-349, 362 when an error is detected on their inputs. The formula for
 5 calculating each output error is as aforementioned.

The register components 360 can be defined to be of a number of types in accordance with requirements as previously discussed when describing the structure of the register set with reference to Table 5.

3.14 CBus Structure

10 As noted previously, the CBus (control bus) is responsible for the overall control of each module by way transferring information for the setting of registers within each module's standard CBus interface. It will be evident from the description of the standard CBus interface that the CBus serves two main purposes:

- 15 1. It is the control bus that drives each of the modules.
2. It is the access bus for RAMs, FIFOs and status information contained within each of the modules.

The CBus uses an instruction-address-data protocol to control modules by the setting configuration registers within the modules. In
 20 general, registers will be set on a per instruction basis but can be modified at any time. The CBus gathers status and other information, and accesses RAM and FIFO data from the various modules by requesting data.

The CBus is driven on a transaction by transaction basis either by:

1. the Instruction Controller 235 (Fig. 2) when executing
 25 instructions,
2. the External Interface Controller 238 (Fig. 2) when performing a target (slave) mode bus operation, or
3. an external device if the External CBus Interface is so configured.

In each of these cases, the driving module is considered to be the source module of the CBus, and all other modules possible destinations. Arbitration on this bus is carried out by the Instruction Controller.

The following table sets out one form of CBus signal definitions
 5 suitable for use with the preferred embodiment:

Table 6: CBus Signal Definition

Name	Type	Definition
c_iad[31:0]	source	instruction-address-data
c_valid	source	CBus instruction valid
c_sdata[31:0]	destination	status/read data
c_svalid	destination	status/read data valid
c_reset[15:0]	source	reset lines to each module
c_active[15:0]	destination	active lines from each module
c_background[15:0]	destination	background active lines from each module
c_int[15:0]	destination	interrupt lines from each module
c_error[15:0]	destination	error lines from each module
c_req1, c_req2	EIC, external	bus control request

Name	Type	Definition
c_gnt1, c_gnt2	IC	bus control grant
c_end	IC	end of instruction
clk	global	clock

A CBus `c_iad` signal contains the addressing data and is driven by the controller in two distinct cycles:

1. Instruction cycles (`c_valid` high) where the CBus instruction and an address is driven onto `c_iad`; and
- 5 2. Data cycles (`c_valid` low) where data is driven onto `c_iad` (write operations) or `c_sdata` (read operations).

In the case of a write operation, the data associated with an instruction is placed on the `c_iad` bus in the cycle directly following the instruction cycle. In the case of a read operation, the target module of the
 10 read operation drives the `c_sdata` signal until the data cycle completes.

Turning now to Fig. 21, the bus includes a 32 bit instruction-address-data field which can be one of three types 370-372:

1. Type A operations (370) are used to read and write registers and the per-module data areas within the co-processor. These operations can
 15 be generated by the external interface controller 238 performing target mode PCI cycles, by the instruction controller 231 configuring the co-processor for an instruction, and by the External CBus Interface.

For these operations, the data cycle occurs in the clock cycle immediately following the instruction cycle. The data cycle is acknowledged
 20 by the designation module using the `c_svalid` signal.

2. Type B operations (371) are used for diagnostic purposes to access any local memory and to generate cycles on the Generic Interface. These operations will be generated by the External Interface Controller performing target mode PCI cycles and by the External CBus Interface. The
 25 data cycle can follow at any time after the instruction cycle. The data cycle is acknowledged by the destination module using the `c_svalid` signal.

3. Type C operations (372) are used to set individual bits within a module's registers. These operations will be generated by the instruction controller 231 configuring the co-processor's for an instruction and by the

External CBus Interface. There is no data cycle associated with a Type C operation, data is encoded in the instruction cycle.

The type field of each instruction encodes the relevant CBus transaction type in accordance with the following table:

5

Table 7: CBus Transaction Types

c_iad.type value	transaction type	instruction format type
0000	no-op	A, B, C
0001	reserved	
0010	peripheral interface write	B
0011	peripheral interface read	B
0100	generic bus write	B
0101	generic bus read	B
0110	local memory write	B
0111	local memory read	B
1000	register write	A
1001	register read	A
1010	module memory write	A
1011	module memory read	A
1100	test mode write	A
1101	test mode read	A
1110	bit set	C
1111	reserved	

The byte field is utilized for enabling bits within a register to be set. The module field sets out the particular module to which an instruction on the CBus is addressed. The register field sets out which of the registers within a module is to be updated. The address field is utilized for addressing memory portions where an operation is desired on those memory portions and can be utilized for addressing RAMs, FIFOs, etc. The enable field enables selected bits within a selected byte when a bit set instruction is utilized. The data field contains the bit wise data of the bits to be written to the byte selected for update.

As noted previously, the CBus includes a `c_active` line for each module, which is asserted whenever a module has outstanding activity pending. The instruction controller utilizes these signals to determine when an instruction has completed. Further, the CBus contains a `c_background` line for each module that can operate in a background mode in addition to any preset, error and interrupt lines, one for each module, for resetting, detecting errors and interrupts.

3.15 Co-processor Data Types and Data Manipulation

Returning now to Fig. 2, in order to substantially simplify the operation of the co-processor unit 224, and in particular the operation of the major computational units within the co-processor being the JPEG coder 241 and the main data path 242, the co-processor utilizes a data model that differentiates between external formats and internal formats. The external data formats are the formats of data as it appears on the co-processor's external interfaces such as the local memory interface or the PCI bus. Conversely, the internal data formats are the formats which appear between the main functional modules of the co-processor 224. This is illustrated schematically in Fig. 22 which shows the various input and output formats. The input external format 381 is the format which is input to the pixel organizer 246, the operand organizer B 247 and the operand organizer C 248. These organizers are responsible for reformatting the input external format data into any of a number of input internal formats 382, which may be inputted to the JPEG coder unit 241 and the main data path unit 242. These two functional units output data in any of a number of output internal formats 383, which are converted by the results organizer 249 to any of a number of required output formats 304.

In the embodiment shown, the external data formats can be divided into three types. The first type is a "packed stream" of data which consists of a contiguous stream of data having up to four channels per data quantum, with each channel consisting of one, two, four, eight or sixteen bit samples.

This packed stream can typically represent pixels, data to be turned into pixels, or a stream of packed bits. The co-processor is designed to utilize little endian byte addressing and big endian bit addressing within a byte. In Fig. 23, there is illustrated a first example 386 of the packed stream format. It is assumed that each object 387 is made up of three channels being channel 0, channel 1 and channel 2, with two bits per channel. The layout of data for this format is as indicated 388. In a next example 390 of Fig. 24, a four channel object 395 having eight bits per channel is illustrated 396 with each data object taking up a 32 bit word. In a third example 395 of Fig. 25, one channel objects 396 are illustrated which each take up eight bits per channel starting at a bit address 397. Naturally, the actual width and number of channels of data will vary depending upon the particular application involved.

A second type of external data format is the "unpacked byte stream" which consists of a sequence of 32 bit words, exactly one byte within each word being valid. An example of this format is shown in Fig. 26 and designated 399, in which a single byte 400 is utilized within each word.

A further external data format is represented by the objects classified as an "other" format. Typically, these data objects are large table-type data representing information such as colour space conversion tables, Huffman coding tables and the like.

The co-processor utilizes four different internal data types. A first type is known as a "packed bytes" format which comprizes 32 bit words, each consisting of four active bytes, except perhaps for a final 32 bit word. In Fig. 27, there is illustrated one particular example 402 of the packed byte format with 4 bytes per word.

The next data type, illustrated with reference to Fig. 28, is "pixel" format and comprises 32 bit words 403, consisting of four active byte channels. This pixel format is interpreted as four channel data.

A next internal data type illustrated with reference to Fig. 29 is an "unpacked byte" format, in which each word consists of one active byte channel

405 and three inactive byte channels, the active byte channel being the least significant byte.

All other internal data objects are classified by the "other" data format.

5 Input data in a given external format is converted to the appropriate internal format. Fig. 30 illustrates the possible conversions carried out by the various organizers from an external format 410 to an internal format 411. Similarly, Fig. 31 illustrates the conversions carried out by the results organizer 249 in the conversion from internal formats 412 to external formats 10 413.

The circuitry to enable the following conversions to take place are described in greater detail below.

Turning firstly to the conversion of input data external formats to internal formats, in Fig. 32 there is shown the methodology utilized by the 15 various organizers in the conversion process. Starting initially with the external other format 416, this is merely passed through the various organizers unchanged. Next, the external unpacked byte format 417 undergoes unpacked normalization 418 to produce a format 419 known as internally unpacked bytes. The process of unpacked normalization 418 20 involves discarding the three inactive bytes from an externally unpacked byte stream. The process of unpacked normalization is illustrated in Fig. 33 wherein the input data 417 having four byte channels wherein only one byte channel is valid results in the output format 419 which merely comprizes the bytes themselves.

25 Turning again to Fig. 32, the process of packed normalization 421 involves translating each component object in an externally packed stream 422 into a byte stream 423. If each component of a channel is less than a byte in size then the samples are interpolated up to eight bit values. For example, when translating four bit quantities to byte quantities, the four bit 30 quantity 0xN is translated to the byte value 0xNN. Objects larger than one

byte are truncated. The input object sizes supported on the stream 422 are 1, 2, 4, 8 and 16 bit sizes, although again these may be different depending upon the total width of the data objects and words in any particular system to which the invention is applied.

5 Turning now to Fig. 34, there is illustrated one form of packed normalization 421 on input data 422 which is in the form of 3 channel objects with two bits per channel (as per the data format 386 of Fig. 23). The output data comprizes a byte channel format 423 with each channel "interpolated up" where necessary to comprize an eight bit sample.

10 Returning to Fig. 32, the pixel streams are then subjected to either a pack operation 425, an unpacked operation 426 or a component selection operation 427.

In Fig. 35 there is shown an example of the packed operation 425 which simply involves discarding the inactive byte channel and producing a
15 byte stream, packed up with four active bytes per word. Hence, a single valid byte stream 430 is compressed into a format 431 having four active bytes per word. The unpacking operation 426 involves almost the reverse of the packing operation with the unpacked bytes being placed in the least significant byte of a word. This is illustrated in Fig. 36 wherein a packed
20 byte stream 433 is unpacked to produce result 434.

The process of component selection 427 is illustrated in Fig. 37 and involves selecting N components from an input stream, where N is the number of input channels per quantum. The unpacking process can be utilized to produce "prototype pixels" eg. 437, with the pixel channels filled from the
25 least significant byte. Turning to Fig. 38, there is illustrated an example of component selection 440 wherein input data in the form 436 is transformed by the component selection unit 427 to produce prototype pixel format 437.

After component selection, a process of component substitution 440 (Fig. 32) can be utilized. The component substitution process 440 is
30 illustrated in Fig. 38 and comprizes replacing selected components with a

constant data value stored within an internal data register 441 to produce, as an example, output components 242.

Returning again to Fig. 32, the output of stages 425, 426 and 440 is subjected to a lane swapping process 444. The lane swapping process, as illustrated in Fig. 39, involves a byte-wise multiplexing of any lane to any other lane, including the replication of a first lane onto a second lane. The particular example illustrated in Fig. 39 includes the replacement of channel 3 with channel 1 and the replication of channel 3 to channels 2 and channel 1.

Returning again to Fig. 32, after the lane swapping step 444 the data stream can be optionally stored in the multi-used value RAM 250 before being read back and subjected to a replication process 446.

The replication process 446 simply replicates the data object whatever it may be. In Fig. 40, there is illustrated a process of replication 446 as applied to pixel data. In this case, the replication factor is one.

In Fig. 41, there is illustrated a similar example of the process of replication applied to packed byte data.

In Fig. 42, there is illustrated the process utilized by the result organizer 249 for transferral of data in an output internal format 383 to an output external format 384. This process includes equivalent steps 424, 425, 426 and 440 to the conversion process described in Fig. 32. Additionally, the process 450 includes the steps of component deselection 451, denormalization 452, byte addressing 453 and write masking 454. The component deselection process 451, as illustrated in Fig. 43, is basically the inverse operation of the component selection process 427 of Fig. 37 and involves the discarding of unwanted data. For example, in Fig. 43, only 3 valid channels of the input are taken and packed into data items 456.

The denormalization process 452 is illustrated with reference to Fig. 44 and is loosely the inverse operation of the packed normalization process 421 of Fig. 34. The denormalization process involves the translation of each object or data item, previously treated as a byte, to a non-byte value.

The byte addressing process 453 of Fig. 42 deals with any byte wize reorganization that is necessary to deal with byte addressing issues. For an externally unpacked byte output stream, the least two significant bits of the stream's address correspond to the active stream. The byte addressing step 5 453 is responsible for re-mapping the output stream from one byte channel to another when external unpacked bytes are utilized (Fig. 45). Where an externally packed stream is utilized (Fig. 46), the byte addressing module 453 remaps the start address of the output stream as illustrated.

The write masks process 454 of Fig. 42 is illustrated in Fig. 47 and is 10 used to mask off a particular channel eg. 460 of a packed stream which is not to be written out.

The details of the input and output data type conversion to be applied are specified by the contents of the corresponding Data Manipulation Registers:

- 15 • The Pixel Organizer Data Manipulation Register (po_dmr)
- The Operand Organizer B and Operand Organizer C Data Manipulation Registers (oob_dmr, ooc_dmr);
- The Result Organizer Data Manipulation Register (ro_dmr);

Each of the Data Manipulation Registers can be set up for an 20 instruction in one of two ways:

1. They can be explicitly set using any of the standard methods for writing to the co-processor's registers immediately prior to the execution of the instruction; or
2. They can be set up by the co-processor itself to reflect a current 25 instruction.

During the instruction decoding process, the co-processor examines the contents of the Instruction Word and the Data Word of the instruction to determine, amongst other things, how to set up the various Data Manipulation Registers. Not all combinations of the instruction and 30 operands make sense. Several instructions have implied formats for some

operands. Instructions that are coded with inconsistent operands may complete without error, although any data so generated is "undefined". If the 'S' bit of the corresponding Data Descriptor is 0, the co-processor sets the Data Manipulation Register to reflect the current instruction.

- 5 The format of the Data Manipulation Registers is illustrated in Fig. 48. The following table sets out the format of the various bits within the registers as illustrated in Fig. 48:

Table 8: Data Manipulation Register Format

Field	Description
ls3	Lane Swap for byte 3 (most significant byte)
ls2	Lane swap for byte 2
ls1	Lane swap for byte 1
ls0	Lane swap for byte 0
suben	Substitution Enables 1 = substitute data from Internal Data Register for this byte 0 = do not substitute data from Internal Data Register for this byte
replicate	Replication Count Indicates the number of <i>additional</i> data items to generate.
wrmask	Write Masks 0 = write out corresponding byte channel 1 = do not write out corresponding byte channel
cmsb	Choose most significant bits 0 = choose least significant bits of a byte when performing denormalization (useful for halftoning operations) 1 = choose most significant bits of a byte when performing denormalization (useful as inverse of input normalization)

Field	Description
normalize	<p>Normalization factor: represents the number of bits to be translated to a byte:</p> <p>0 = 1 bit data objects</p> <p>1 = 2 bit data objects</p> <p>2 = 4 bit data objects</p> <p>3 = 8 bit data objects</p> <p>4 = 16 bit data objects</p>
bo	<p>Bit Offset: represents the starting bit address for objects smaller than a byte. Bit addressing is big endian.</p>
P	<p>External Format:</p> <p>0 = unpacked bytes</p> <p>1 = packed stream</p>
if	<p>Internal Format:</p> <p>0 = pixels</p> <p>1 = unpacked bytes</p> <p>2 = packed bytes</p> <p>3 = other</p>
∞	<p>Channel count:</p> <p>For the Input Organizers this defines the number of normalized input bytes collected to form each internal data word during component selection. For the Output Organizer this defines the number of valid bytes from the internal data word that will be used to construct output data.</p> <p>0 = 4 active channels</p> <p>1 = 1 active channels</p> <p>2 = 2 active channels</p> <p>3 = 3 active channels</p>
L	<p>Immediate data:</p> <p>0 = not long: immediate data</p> <p>1 = long: pointer to data</p>

Field	Description
what	addressing mode: 0 = instruction specific mode 1 = sequential addressing 2 = tile addressing 3 = constant data. ie, one item of internal data is produced, and this item is used repetitively.

A plurality of internal and external data types may be utilized with each instruction. All operand, results and instruction type combinations are potentially valid, although typically only a subset of those combinations will lead to meaningful results. Particular operand and result data types that are expected for each instruction are detailed below in a first table (Table 9) summarising the expected data types for external and internal formats:

Table 9: Expected Data Types

Instruction	Operand A (Pixel Organizer)		Operand B (Operand Organizer B)		Operand C (Operand Organizer C)		Result (Result Organizer)	
Compositing	ps	px	ps	px(T) bl(B)	ps ub const	ub	px ub	ps ub
GCSC	ps ift	ift	mcsc scsc (B)	mcsc scsc (B)	mcsc scsc (B)	mcsc scsc (B)		
JPEG comp.	ps us	pb	et (B)	et (B)	et (B)	et (B)	ub	ps
JPEG decomp	ps	pb	fdt sdt (B)	fdt sdt (B)	fdt sdt (B)	fdt sdt (B)	pb	ps ub
Data coding	ps ub	px pb ub	et fdt sdt (B)	et fdt sdt (B)	et fdt sdt (B)	et fdt sdt (B)	px pb ub	ps ub
Transformations and Convolutions	skd lkd	skd lkd	it (B)	it (B)	it (B)	it (B)	px	ps ub
Matrix Multiplication	ps ub	px	mm (B)	mm (B)	mm (B)	mm(B)	px	ps ub

Instruction	Operand A (Pixel Organizer)		Operand B (Operand Organizer B)		Operand C (Operand Organizer C)		Result (Result Organizer)	
	ps ub	px pb ub	ps ub	px pb ub	-	-	px pb ub	ps ub
Halftoning	ps ub	px pb ub	ps ub	px pb ub	-	-	px pb ub	ps ub
Hierarchical Image: horizontal interpolation	ps ub	px pb ub	-	-	-	-	px pb ub	ps ub
Hierarchical Image: vertical interpolation and residual merging	ps ub	px pb ub	ps ub	px pb ub	-	-	px pb ub	ps ub
General Memory Copy	ps ub	px pb ub	-	-	-	-	px pb ub	ps ub
Peripheral DMA	-	-	-	-	-	-	-	-
Internal Access	-	-	-	-	-	-	-	-
Flow Control	-	-	-	-	-	-	-	-

The symbols utilized in the above table are as follows:

Table 10: Symbol Explanation

Symbol	Explanation
ps	packed stream
pb	packed bytes
ub	unpacked bytes
px	pixels
bl	blend
const	constant
mesc	4 output channel
scsc	1 output channel color conversion table
ift	Interval and Fraction tables
et	JPEG encoding table
fdt	fast JPEG decoding table
sdt	slow JPEG decoding table
skd	short kernel descriptor
lkd	long kernel descriptor
mm	matrix co-efficient table

Symbol	Explanation
it	image table
(B)	this organizer in bypass mode for this operation
(T)	operand may tile
-	no data flows via this operand

3.16 Data Normalization Circuit

5 Referring to Fig. 49, there is shown a computer graphics processor having three main functional blocks: a data normalizer 1062 which may be implemented in each of the pixel organizer 246 and operand organizers B and C 247, 248, a central graphics engine in the form of the main data path 242 or JPEG units 241 and a programming agent 1064, in the form of an instruction

10 controller 235. The operation of the data normalizer 1062 and the central graphics engine 1064 is determined by an instruction stream 1066 that is provided to the programming agent 1064. For each instruction, the programming agent 1064 performs a decoding function and outputs internal control signals 1067 and 1068 to the other blocks in the system. For each

15 input data word 1069, the normalizer 1062 will format the data according to

the current instruction and pass the result to the central graphics engine 1063, where further processing is performed.

The data normalizer represents, in a simplified form, the pixel organizer and the operand organizers B and C. Each of these organizers
 5 implements the data normalization circuitry, thereby enabling appropriate normalization of the input data prior to it passing to the central graphics engine in the form of the JPEG coder or the main data path.

The central graphics engine 1063 operates on data that is in a standard format, which in this case is 32-bit pixels. The normalizer is thus
 10 responsible for converting its input data to a 32-bit pixel format. The input data words 1069 to the normalizer are also 32 bits wide, but may take the form of either packed components or unpacked bytes. A packed component input stream consists of consecutive data objects within a data word, the data objects being 1,2,4,8 or 16 bits wide. By contrast, an unpacked byte input
 15 stream consists of 32-bit words of which only one 8-bit byte is valid. Furthermore, the pixel data 11 produced by the normalizer may consist of 1,2,3 or 4 valid channels, where a channel is defined as being 8 bits wide.

Turning now to Fig. 50, there is illustrated in greater detail a particular hardware implementation of the data normalizer 1062. The data
 20 normalization unit 1062 is composed of the following circuits: a First-In-First-Out buffer (FIFO) 1073, a 32-bit input register (REG1) 1074, a 32-bit output register (REG2) 1076, normalization multiplexors 1075 and a control unit 1076. Each input data word 1069 is stored in the FIFO 1073 and is subsequently latched into REG1 1074, where it remains until all its input
 25 bits have been converted into the desired output format. The normalization multiplexors 1075 consist of 32 combinatorial switches that produce pixels to be latched into REG2 by selecting bits from the value in REG1 1074 and the current output of the FIFO 1073. Thus the normalization multiplexors 1075 receive two 32-bit input words 1077, 1078, denoted as $x[63..32]$ and $x[31..0]$.

It has been found that such a method improves the overall throughput of the apparatus, especially when the FIFO contains at least two valid data words during the course of an instruction. This is typically due to the way in which data words originally fetched from memory. In some cases, a desired
 5 data word or object may be spread across or "wrapped" into a pair of adjacent input data words in the FIFO buffer. By using an additional input register 1074, the normalization multiplexers can reassemble a complete input data word using components from adjacent data words in the FIFO buffer, thereby avoiding need for additional storage or bit-stripping operations prior to the
 10 main data manipulation stages. This arrangement is particularly advantageous where multiple data words of a similar type are inputted to the normalizer.

The control unit generates enable signals REG1_EN 20 and REG2_EN[3..0] 1081 for updating REG1 1074 and REG2 1076, respectively,
 15 as well as signals to control the FIFO 1073 and normalization multiplexors 1075.

The programming agent 1064 in Fig. 49 provides the following configuration signals for the data normalizer 1062: a FIFO_WR 4 signal, a normalization factor $n[2..0]$, a bit offset $b[2..0]$, a channel count $c[1..0]$ and an
 20 external format (E). Input data is written into the FIFO 1073 by asserting the FIFO_WR signal 1085 for each clock cycle that valid data is present. The FIFO asserts a `fifo_full` status flag 1086 when there is no space available. Given 32-bit input data, the external format signal is used to determine whether the input is in the format of a packed stream (when $E=1$) or consists
 25 of unpacked bytes (when $E=0$). For the case when $E=1$, the normalization factor encodes the size of each component of a packed stream, namely: $n=0$ denotes 1-bit wide components, $n=1$ denotes 2 bits per component, $n=2$ denotes 4 bits per component, $n=3$ denotes 8-bit wide components and $n>3$ denotes 16-bit wide components. The channel count encodes the maximum
 30 number of consecutive input objects to format per clock cycle in order to

produce pixels with the desired number of valid bytes. In particular, $c=1$ yields pixels with only the least significant byte valid, $c=2$ denotes least significant 2 bytes valid, $c=3$ denotes least significant 3 bytes valid and $c=0$ denotes all 4 bytes valid.

- 5 When a packed stream consists of components that are less than 8 bits wide, the bit offset determines the position in $x[31..0]$, the value stored in REG1, from which to begin processing data. Assuming a bit offset relative to the most significant bit of the first input byte, the method for producing an output data byte $y[7..0]$ is described by the following set of equations :-

10

where $n = 0$:

$$y[i] = x[7-b], \quad \text{where } 0 \leq i \leq 7$$

where $n = 1$:

15 $y[i] = x[7-b], \quad \text{where } i = 1, 3, 5, 7$

$$y[i] = x[6-b], \quad \text{where } i = 0, 2, 4, 6$$

where $n = 2$:

$$y[3] = x[7-b]$$

20 $y[2] = x[6-b]$

$$y[1] = x[5-b]$$

$$y[0] = x[4-b]$$

$$y[7] = y[3]$$

$$y[6] = y[2]$$

25 $y[5] = y[1]$

$$y[4] = y[0]$$

where $n = 3$:

$$y[i] = x[i], \quad \text{where } 0 \leq i \leq 7$$

30

where $n > 3$:

$$y[7..0] = x[15..8]$$

Corresponding equations may be used to generate output data bytes
 5 $y[15..8]$, $y[23..16]$ and $y[31..24]$.

The above method may be generalized to produce an output array of any length by taking each component of the input stream and replicating it as many times as necessary to generate output objects of standard width. In addition, the order of processing each input component may be defined as
 10 little-endian or big-endian. The above example deals with big-endian component ordering since processing always begins from the most significant bit of an input byte. Little-endian ordering requires redefinition of the bit offset to be relative to the least significant bit of an input byte. In situations where the input component width exceeds the standard output width, output
 15 components are generated by truncating each input component, typically by removing a suitable number of the least significant bits. In the above set of equations, truncation of 16-bit input components to form 8-bit wide standard output is performed by selecting the most significant byte of each 16-bit data object.

20 The control unit of Fig. 50 performs the decoding of $n[2..0]$ and $c[1..0]$, and uses the result along with $b[2..0]$ and E to provide the select signals for the normalization multiplexors and the enable signals for REG1 and REG2. Since the FIFO may become empty during the course of an instruction, the control unit also contains counters that record the current bit position,
 25 $in_bit[4..0]$, in REG1 from which to select input data, and the current byte, $out_byte[1..0]$, in REG2 to begin writing output data. The control unit detects when it has completed processing each input word by comparing the value of $in_bit[4..0]$ to the position of the final object in REG1, and initiates a FIFO read operation by asserting the $FIFO_RD$ signal for one clock cycle when
 30 the FIFO is not empty. The signals $fifo_empty$ and $fifo_full$ denote the FIFO

status flags, such that `fifo_empty=1` when the FIFO contains no valid data, and `fifo_full=1` when the FIFO is full. In the same clock cycle that `FIFO_RD` is asserted, `REG1_EN` is asserted so that new data are captured into `REG1`. There are 4 enable signals for `REG2`, one for each byte in the output register.

5 The control unit calculates `REG2_EN[3..0]` by taking the minimum of the following 3 values : the decoded version of `c[1..0]`, the number of valid components remaining to be processed in `REG1`, and the number of unused channels in `REG2`. When `E=0` there is only one valid component in `REG1`. A complete output word is available when the number of channels that have

10 been filled in `REG2` is equal to the decoded version of `c[1..0]`.

In a particularly preferred embodiment of the invention, the circuit area occupied by the apparatus in Fig. 50 can be substantially reduced by applying a truncation function to the bit offset parameter, such that only a restricted set of offsets are used by the control unit and normalization

15 multiplexors. The offset truncation depends upon the normalization factor and operates according to the following equation:

$$\begin{aligned}
 b_trunc[2..0] &= 0, \quad \text{where } n \geq 3 \\
 &= b[2..0], \quad \text{where } n = 0 \\
 &= b[2..1] \& "0", \quad \text{where } n = 1 \\
 20 \quad &= b[2] \& "00", \quad \text{where } n = 2
 \end{aligned}$$

(Note that "&" denotes bitwise concatenation).

The above method allows each of the normalization multiplexors, denoted in Fig. 50 by `MUX0`, `MUX1` ... `MUX31`, to be reduced from 32-to-1 in

25 size when no truncation is applied, to be a maximum size of 20-to-1 with bit offset truncation. The size reduction in turn leads to an improvement in circuit speed.

It can be seen from the foregoing that the preferred embodiment provides an efficient circuit for the transformation of data into one of a few

30 normalized forms.

3.17 Image Processing Operations of Accelerator Card

Returning again to Fig. 2 and Table 2, as noted previously, the instruction controller 235 "executes" instructions which result in actions being performed by the co-processor 224. The instructions executed include a number of instructions for the performance of useful functions by the main data path unit 242. A first of these useful instructions is compositing.

3.17.1 Compositing

Referring now to Fig. 51, there is illustrated the compositing model implemented by the main data path unit 242. The compositing model 462 generally has three input sources of data and the output data or sink 463. The input sources can firstly include pixel data 464 from the same destination within the memory as the output 463 is to be written to. The instruction operands 465 can be utilized as a data source which includes the color and opacity information. The color and opacity can be either flat, a blend, pixels or tiled. The flat or blend is generated by the blend generator 467, as it is quicker to generate them internally than to fetch via input/output. Additionally, the input data can include attenuation data 466 which attenuates the operand data 465. The attenuation can be flat, bit map or a byte map.

As noted previously, pixel data normally consists of four channels with each channel being one byte wide. The opacity channel is considered to be the byte of highest address. For an introduction to the operation and usefulness of compositing operations, reference is made to the standard texts including the seminal paper by Thomas Porter and Tom Duff "Compositing Digital Images" in Computer Graphics, Volume 18, Number 3, July 1984.

The co-processor can utilize pre-multiplied data. Pre-multiplication can consist of pre-multiplying each of the colored channels by the opacity channel. Hence, two optional pre-multiplication units 468, 469 are provided for pre-multiplying the opacity channel 470, 471 by the colored data to form, where required, pre-multiplied outputs 472, 473. A compositing unit 475

implements a composite of its two inputs in accordance with the current instruction data. The compositing operators are illustrated in Table 11 below:

5

Table 11: Compositing Operations

Operator	Definition
(a_{co}, a_o) over (b_{co}, b_o)	$(a_{co} + b_{co}(1 - a_o), a_o + b_o(1 - a_o))$
(a_{co}, a_o) in (b_{co}, b_o)	$(a_{co}b_o, a_ob_o)$
(a_{co}, a_o) out (b_{co}, b_o)	$(a_{co}(1 - b_o), a_o(1 - b_o))$
(a_{co}, a_o) atop (b_{co}, b_o)	$(a_{co}b_o + b_{co}(1 - a_o), b_o)$

Operator	Definition
(a_{co}, a_o) xor (b_{co}, b_o)	$(a_{co}(1 - b_o) + b_{co}(1 - a_o), a_o(1 - b_o) + b_o(1 - a_o))$
(a_{co}, a_o) plus (b_{co}, b_o)	$(wc(a_{co} + b_{co} - r(a_o + b_o - 255)/255) + r(\text{clamp}(a_o + b_o) - 255)/255, \text{clamp}(a_o + b_o))$
(a_{co}, a_o) loadzero (b_{co}, b_o)	$(0, 0)$
(a_{co}, a_o) loadc (b_{co}, b_o)	(b_{co}, a_o)
(a_{co}, a_o) loado (b_{co}, b_o)	(a_{co}, b_o)
(a_{co}, a_o) loadco (b_{co}, b_o)	(b_{co}, b_o)

The nomenclature (a_{co}, a_o) refers to a pre-multiplied pixel of color a_c and opacity a_o . R is an offset value and "wc" is a wrapping/clamping operator whose operation is explained below. It should be noted that the reverse operation of each operator in the above table is also implemented by a compositing unit 475.

A clamp/wrapping unit 476 is provided to clamp or wrap data around the limit values 0-255. Further, the data can be subjected to an optional "unpre-multiplication" 477 restoring the original pixel values as required. Finally, output data 463 is produced for return to the memory.

In Fig. 52, there is illustrated the form of an instruction word directed to the main data path unit for compositing operations. When the X field in

the major op-code is 1, this indicates a plus operator is to be applied in accordance with the aforementioned table. When this field is 0, another instruction apart from the plus operator is to be applied. The P_a field determines whether or not to pre-multiply the first data stream 464 (Fig. 51).

5 The P_b field determines whether or not to pre-multiply the second data stream 465. The P_r field determines whether or not to "unpre-multiply" the result utilising unit 477. The C field determines whether to wrap or clamp, overflow or underflow in the range 0-255. The "com-code" field determines which operator is to be applied. The plus operator optionally utilizes an

10 offset register (mdp_por). This offset is subtracted from the result of the plus operation before wrapping or clamping is applied. For plus operators, the com-code field is interpreted as a per channel enablement of the offset register.

The standard instruction word encoding 280 of Fig. 10 previously

15 discussed is altered for composting operands. As the output data destination is the same as the source, operand A will always be the same operand as the result word so operand A can be utilized in conjunction with operand B to describe at greater length the operand B. As with other instructions, the A descriptor within the instructions still describes the

20 format of the input and the R descriptor defines the format of the output.

Turning now to Fig. 53, there is illustrated in a first example 470, the instruction word format of a blend instruction. A blend is defined to have a start 471 and end value 472 for each channel. Similarly, in Fig. 54 there is

25 illustrated 475 the format of a tile instruction which is defined by a tile address 476 a start offset 477, a length 478. All tile addresses and dimensions are specified in bytes. Tiling is applied in a modular fashion and, in Fig. 55, there is shown the interpretation of the fields 476 - 478 of Fig. 54. The tile address 476 denotes the start address in memory of the tile. A tile start offset 477 designates the first byte to be utilized as a start of the tile.

30 The tile length 478 designates the total length of the tile for wrap around.

Returning to Fig. 51, every color component and opacity can be attenuated by an attenuation value 466. The attenuation value can be supplied in one of three ways:

1. Software can specify a flat attenuation by placing the attenuation factor in the operand C word of the instruction.
2. A bit map attenuation where 1 means fully on and 0 means fully off can be utilized with software specifying the address of the bit map in the operand C word of the instruction.
3. Alternatively, a byte map attenuation can be provided again with the address of the byte map in operand C.

Since the attenuation is interpreted as an unsigned integer from 0-255, the pre-multiplied color channel is multiplied by the attenuation factor by effectively calculating:

$$C_{oa} = C_{oa} \times A/255$$

Where A is the attenuation and C_o is the pre-multiplied color channel.

3.17.2 Color Space Conversion Instructions

Returning again to Fig. 2 and Table 2, the main data path unit 242 and data cache 230 are also primarily responsible for color conversion. The color space conversion involves the conversion of a pixel stream in a first color space format, for example suitable for RGB color display, to a second color space format, for example suitable for CYM or CYMK printing. The color space conversion is designed to work for all color spaces and can be used for any function from at least one to one or more dimensions.

The instruction controller 235 configures, via the Cbus 231, the main data path unit 242, the data cache controller 240, the input interface switch 252, the pixel organizer 246, the MUV buffer 250, the operand organizer B 247, the operand organizer C 248 and the result organizer 249 to operate in the color conversion mode. In this mode, an input image consisting of a plurality of lines of pixels is supplied, one line of pixels after another, to the main data path unit 242 as a stream of pixels. The main data path unit 242

(Fig. 2) receives the stream of pixels from the input interface switch 252 via the pixel organizer 246 for color space conversion processing one pixel at a time. In addition, interval and fractional tables are pre-loaded into the MUV buffer 250 and color conversion tables are loaded into the data cache 230. The main data path unit 242 accesses these tables via the operand organizers B and C, and converts these pixels, for example from the RGB color space to the CYM or CYMK color space and supplies the converted pixels to the result organizer 249. The main data path unit 242, the data cache 230, the data controller 240 and the other abovementioned devices are able to operate in either of the following two modes under control of the instruction controller 235; a Single Output General Color Space (SOGCS) Conversion mode or a Multiple Output General Color Space (MOGCS) Conversion Mode. For more details on the data cache controller 240 and data cache 230, reference is made to the section entitled *Data Cache Controller and Cache* 240, 230 (Fig. 2).

Accurate color space conversion can be a highly non-linear process. For example, color space conversion of a RGB pixel to a single primary color component (e.g. cyan) of the CYMK color space is theoretically linear, however in practice non-linearities are introduced typically by the output device which is used to display the colour components of the pixel. Similarly for the color space conversion of the RGB pixel to the other primary color components (yellow, magenta or black) of the CYMK color space. Consequently a non-linear colour space conversion is typically used to compensate for the non-linearities introduced on each colour component. The highly non-linear nature of the color conversion process requires either a complex transfer function to be implemented or a look-up table to be utilized. Given an input color space of, for example, 24 bit RGB pixels, a look-up table mapping each of these pixels to a single 8 bit primary color component of the CYMK color space (i.e. cyan) would require over 16 megabytes. Similarly, a look-up table simultaneously mapping the 24 bit RGB pixels to all four 8 bit primary color

components of the CYMK color space would require over 64 megabytes, which is obviously excessive. Instead, the main data path 242 (Fig. 2) uses a look-up table stored in the data cache 230 having sparsely located output color values corresponding to points in the input color space and interpolates
 5 between the output color values to obtain an intermediate output.

a. Single Output General Color Space (SOGCS) Conversion Mode

In both the single and multiple output color conversion modes (SOGCS) and (MOGCS), the RGB color space is comprized of 24 bit pixels having 8 bit red, green and blue color components. Each of the RGB
 10 dimensions of the RGB color space is divided into 15 intervals with the length of each interval having a substantially inverse proportionality to the non-linear behavior of the transfer function between the RGB to CYMK color space of the printer. That is, where the transfer function has a highly non-linear behavior the interval size is reduced and where the transfer function has a
 15 more linear behavior, the size of the interval is increased. Preferably, the color space of each output printer is accurately measured to determine those non-linear portions of its transfer function. However, the transfer function can be approximated or modelled based on know-how or measured characteristics of a type printer (e.g.: ink-jet). For each color channel of an
 20 input pixel, the color component value defines a position within one of the 15 intervals. Two tables are used by the main data path unit 242 to determine which interval a particular input color component value lies within and also to determine a fraction along the interval in which a particular input color component value lies. Of course, different tables may be used for output
 25 printers having different transfer functions.

As noted previously, each of the RGB dimensions is divided into 15 intervals. In this way the RGB color space forms a 3-dimensional lattice of intervals and the input pixels at the ends of the intervals form sparsely located points in the input color space. Further, only the output color values
 30 of the output color space corresponding to the endpoints of the intervals are

stored in look-up tables. Hence, an output color value of an input color pixel can be calculated by determining the output color values corresponding to the endpoints of the intervals within which the input pixel lies and interpolating such output color values utilising the fractional values. This technique
 5 reduces the need for large memory storage.

Turning now to Fig. 56, there is illustrated 480 an example of determining for a particular input RGB color pixel, the corresponding interval and fractional values. The conversion process relies upon the utilization of an interval table 482 and a fractional table 483 for each 8 bit input color
 10 channel of the 24 bit input pixel. The 8 bit input color component 481, shown in a binary form in Fig. 56 having the example decimal number 4, is utilized as a look-up to each of the interval and fractional tables. Hence, the number of entries in each table is 256. The interval table 482 provides a 4 bit output defining one of the intervals numbered 0 to 14 into which the input color
 15 component value 481 falls. Similarly, the fractional table 483 indicates the fraction within an interval that the input color value component 481 falls. The fractional table stores 8 bit values in the range of 0 to 255 which are interpreted as a fraction of 256. Hence, for an input color value component 481 having a binary equivalent to the decimal value 4, this value is utilized to
 20 look-up the interval table 482 to produce an output value of 0. The input value 4 is also utilized to look-up the fractional table 483 to produce an output value of 160 which designates the fraction $160/256$. As can be seen from the interval and fractional tables 482 and 483, the interval lengths are not equal. As noted previously, the length of the intervals are chosen
 25 according to the non-linear behavior of the transfer function.

As mentioned above, the separate interval and fractional tables are utilized for each of the RGB color components resulting in three interval outputs and three fractional outputs. Each of the interval and fractional tables for each color component are loaded in the MUV buffer 250 (Fig. 2) and
 30 accessed by the main data path unit 242 when required. The arrangement of

the MUV buffer 250 for the color conversion process is as shown in Fig. 57. The MUV buffer 250 (Fig. 57) is divided into three areas 488, 489 and 490, one area for each color component. Each area e.g. 488 is further divided into a 4 bit interval table and a 8 bit fractional table. A 12 bit output 492 is
 5 retrieved by the main data path unit 242 from the MUV buffer 250 for each input color channel. In the example given above of a single input color component having a decimal value 4, the 12 bit output will be 000001010000.

Turning now to Fig. 58, there is illustrated an example of the interpolation process. The interpolation process consists primarily of
 10 interpolation from one three dimensional space 500, for example RGB color space to an alternative color space, for example CMY or CMYK. The pixels P0 to P7 form sparsely located points in the RGB input color space and having corresponding output color values CV(P0) to CV(P7) in the output color space. The output color component value corresponding to the input pixel Pi falling
 15 between the pixels P0 to P7 is determined by; firstly, determining the endpoints P0, P1,...,P7 of the intervals surrounding the input pixel Pi; secondly, determining the fractional components frac_r, frac_g and frac_b; and lastly interpolating between the output color values CV(P0) to CV(P7) corresponding to the endpoints P0 to P7 using the fractional components.

20 The interpolation process includes a one dimensional interpolation in the red (R) direction to calculate the values temp 11, temp 12, temp 13, temp 14 in accordance with the following equations:

$$\begin{aligned}
 \text{temp 11} &= \text{CV(P0)} + \text{frac_r} (\text{CV(P1)} - \text{CV(P0)}) \\
 \text{temp 12} &= \text{CV(P2)} + \text{frac_r} (\text{CV(P3)} - \text{CV(P2)}) \\
 \text{temp 13} &= \text{CV(P4)} + \text{frac_r} (\text{CV(P5)} - \text{CV(P4)}) \\
 \text{temp 14} &= \text{CV(P6)} + \text{frac_r} (\text{CV(P7)} - \text{CV(P6)})
 \end{aligned}$$

Next, the interpolation process includes the calculation of a further one dimensional interpolation in the green (G) direction utilising the following equations to calculate the values temp 21 and temp 22:

$$\begin{aligned} \text{temp 21} &= \text{temp 11} + \text{frac_g} (\text{temp 12} - \text{temp 11}) \\ \text{temp 22} &= \text{temp 13} + \text{frac_g} (\text{temp 14} - \text{temp 13}) \end{aligned}$$

Finally, the final dimension interpolation in the blue (B) direction is carried out to calculate a final color output value in accordance with the following equation.

$$\text{final} = \text{temp 21} + \text{frag_b} (\text{temp 22} - \text{temp 21})$$

Unfortunately, it is often the case that the input and output gamut may not match. In this respect, the output gamut may be more restricted than the input gamut and in this case, it is often necessary to clamp the gamut at the extremes. This often produces unwanted artefacts when converting using the boundary gamut colors. An example of how this problem can occur will now be explained with reference to Fig. 59, which represents a one dimensional mapping of input gamut values to output gamut values. It is assumed that output values are defined for the input values at points 510 and 511. However, if the greatest output value is clamped at the point 512 then the point 511 must have an output value of this magnitude. Hence, when interpolating between the two points 510 and 511, the line 515 forms the interpolation line and the input point 516 produces a corresponding output value 517. However, this may not be the best color mapping, especially where, without the gamut limitations, the output value would have been at the point 518. The interpolation line between 510 and 518 would produce an output value of 519 for the input point 516. The difference between the two output values 517 and 519 can often lead to unsightly

artefacts, particularly when printing edge of gamut colors. To overcome this problem, the main data path unit can optionally calculate in an expanded output color space and then scale and clamp to the appropriate range utilising the following formula:

$$\text{out} = \begin{cases} 0 & \text{if } x \leq 63 \\ 2(x-64) & \text{if } (64 \leq x \leq 191) \\ 255 & \text{if } (192 \leq x) \end{cases} \quad (4)$$

Returning now to Fig. 58, it will be evident that the interpolation process can either be carried out in the SOCGS conversion mode which converts RGB pixels to a single output color component (for example, cyan) or the MOGCS mode which converts RGB pixels to all the output color components simultaneously. Where color conversion is to be carried out for each pixel in an image, many millions of pixels may have to be independently color converted. Hence, in order for high speed operation, it is desirable to be able to rapidly locate the 8 values (P0-P7) around a particular input value.

As noted previously with respect to Fig. 57, the main data path unit 242 retrieves for each color input channel, a 12 bit output consisting of a 4 bit interval part and a 8 bit fractional part. The main data path unit 242 concatenates these 4 bit interval parts of the red, green and blue color channels to form a single 12 bit address (I_R, I_G, I_B), as shown in Fig. 60 as 520.

Fig. 60 shows a data flow diagram illustrating the manner in which a single output color component 563 is obtained in response to the single 12 bit address 520. The 12 bit address 520 is first fed to an address generator of the data cache controller 240, such as the generator 1881 (shown in Fig. 141) which generates 8 different 9 bit line and byte addresses 521 for memory banks (B_0, B_1, \dots, B_7). The data cache 230 (Fig. 2) is divided into 8 independent memory banks 522 which can be independently addressed by the

respective 8 line and byte addresses. The 12 bit address 520 is mapped by the address generator into the 8 line and byte addresses in accordance with the following table:

5 **Table 12: Address Composition for SOGCS Mode**

	Bit [8:6]	Bit [5:3]	Bit [2:0]
Bank 7	R[3:1]	G[3:1]	B[3:1]
Bank 6	R[3:1]	G[3:1]	B[3:1]+B[0]
Bank 5	R[3:1]	G[3:1]+G[0]	B[3:1]
Bank 4	R[3:1]	G[3:1]+G[0]	B[3:1]+B[0]
Bank 3	R[3:1]+R[0]	G[3:1]	B[3:1]
Bank 2	R[3:1]+R[0]	G[3:1]	B[3:1]+B[0]
Bank 1	R[3:1]+R[0]	G[3:1]+G[0]	B[3:1]
Bank 0	R[3:1]+R[0]	G[3:1]+G[0]	B[3:1]+B[0]

where BIT[8:6], BIT[5:3] and BIT[2:0] represent the sixth to eighth bits, the third to fifth bits and the zero to second bits of the 9 bit bank addresses respectively; and

10 where R[3:1], G[3:1] and B[3:1] represent the first to third bits of the 4 bit intervals I_R , I_G and I_B of the 12 bit address 520 respectively.

Reference is made to memory bank 5 of Table 12 for a more detailed explanation of the 12 bit to 9 bit mapping. In this particular case, the bits 1 to 3 of the 4 bit red interval I_r of the 12 bit address 520 are mapped to bits 6 to 8 of the 9 bit address B5; bits 1 to 3 and bit 0 of the 4 bit green interval I_g are summed and then mapped to bits 3 to 5 of the 9 bit address B5; and bits 1 to 3 of the 4 bit blue interval I_b are mapped to bits 0 to 2 of the 9 bit address B5.

Each of the 8 different line and byte addresses 521 is utilized to
20 address a respective memory bank 522 which consists of 512 x 8 bit entries, and the corresponding 8 bit output color component 523 is latched for each of the memory banks 522. As a consequence of this addressing method, the output color values of CV(P0) to CV(P7) corresponding to the endpoints P0 to

P7 may be located at different positions in the memory banks. For example, a 12 bit address of 0000 0000 0000 will result in the same bank address for each bank, ie 000 000 000. However a 12 bit address of 0000 0000 0001 will result in different bank addresses, ie a bank address of 000 000 000 for banks
 5 7, 5, 3 and 1 and a bank address of 000 000 001 for banks 6, 4, 2 and 0. It is in this way the eight single output color values CV(P0)-CV(P7) surrounding a particular input pixel value are simultaneously retrieved from respective memory banks and duplication of output color values in the memory banks can be avoided.

10 Turning now to Fig. 61, there is illustrated the structure of a single memory bank of the data cache 230 when utilized in the single color conversion mode. Each memory bank consists of 128 line entries 531 which are 32 bits long and comprize 4x8 bit memories 533 - 536. The top 7 bits of the memory address 521 are utilized to determine the corresponding row of
 15 data within the memory address to latch 542 as the memory bank output. The bottom two bits are a byte address and are utilized as an input to multiplexer 543 to determine which of the 4x8 bit entries should be chosen 544 for output. One data item is output for each of the 8 memory banks per clock cycle for return to the main data path unit 242. Hence, the data cache
 20 controller receives a 12 bit byte address from the operand organizer 248 (Fig. 2) and outputs in return to the operand organizers 247, 248, the 8 output color values for interpolation calculation by the main data path unit 242.

Returning to Fig. 60, the interpolation equations are implemented by the main data path unit 242 (Fig. 2) in three stages. In the main data path
 25 unit, a first stage of multiplier and adder units eg. 550 which take as input the relevant color values output by the corresponding memory banks eg. 522 in addition to the red fractional component 551 and calculate the 4 output values in accordance with stage 1 of the abovementioned equations. The outputs eg. 553, 554 of this stage are fed to a next stage unit 556 which
 30 utilizes the frac_g input 557 to calculate an output 558 in accordance with the

aforementioned equation for stage 2 of the interpolation process. Finally, the output 558 in addition to other outputs eg. 559 of this stage are utilized 560 in addition to the frac_b input 562 to calculate a final output color 563 in accordance with the aforementioned equations.

5 The process illustrated in Fig. 60 is implemented in a pipelined manner so as to ensure maximum overall throughput. Further, the method of Fig. 60 is utilized when a single output color component 563 is required. For example, the method of Fig. 60 can be utilized to first produce the cyan color components of an output image followed by the magenta, yellow and
10 black components of an output image reloading the cache tables between passes. This is particularly suitable for a four-pass printing process which requires each of the output colors as part of separate pass.

b. Multiple Output General Color Space Mode

 The co-processor 224 operates in the MOGCS mode in a substantially
15 similar manner to the SOCGS mode, with a number of notable exceptions. In the MOGCS mode, the main data path unit 242, the data cache controller 240 and data cache of Fig. 2 co-operate to produce multiple color outputs simultaneously with four primary colors components being output simultaneously. This would require the data cache 230 to be four times
20 larger in size. However, in the MOGCS mode of operation, in order to save storage space, the data cache controller 240 stores only one quarter of all the output color values of the output color space. The remaining output color values of the output color space are stored in a low speed external memory and are retrieved as required. This particular apparatus and method is
25 based upon the surprising revelation that the implementation of sparsely located color conversion tables in a cache system have an extremely low miss rate. This is based on the insight there is a low deviation in color values from one pixel to the next in most color images. In addition, there is a high probability the sparsely located output color values will be the same for
30 neighboring pixels.

Turning now to Fig. 62 there will now be described the method carried out by the co-processor to implement multi-channel cached color conversion. Each input pixel is broken into its color components and a corresponding interval table value (Fig. 56) is determined as previously described resulting in the three 4 bit intervals I_r , I_g , I_b denoted 570. The combined 12 bit number 570 is utilized in conjunction with the aforementioned table 12 to again derive eight 9-bit addresses. The addresses eg. 572 are then re-mapped as will be discussed below with reference to Fig. 63, and then are utilized to look up a corresponding memory bank 573 to produce four colour output channels 574. The memory bank 573 stores 128x32 bit entries out of a total possible 512x32 bit entries. The memory bank 573 forms part of the data cache 230 (Fig. 2) and is utilized as a cache as will now be described with reference to Fig. 63.

Turning to Fig. 63, the 9 bit bank input 578 is re-mapped as 579 so as to anti-alias memory patterns by re-ordering the bits 580-582 as illustrated. This reduces the likelihood of neighboring pixel values aliasing to the same cache elements.

The reorganized memory address 579 is then utilized as an address into the corresponding memory bank eg. 585 which comprises 128 entries each of 32 bits. The 7 bit line address is utilized to access the memory 585 resulting in the corresponding output being latched 586 for each of the memory banks. Each memory bank, eg 585 has an associated tag memory which comprises 128 entries each of 2 bits. The 7 bit line address is also utilized to access the corresponding tag in tag memory 587. The two most significant bits of the address 579 are compared with the corresponding tag in tag memory 587 to determine if the relevant output color value is stored in the cache. These two most significant bits of the 9 bit address correspond to the most significant bits of the red and green data intervals (see Table 12). Thus in the MOGCS mode the RGB input color space is effectively divided into quadrants along the red and green dimensions where the two most significant

bits of the 9 bit address designates the quadrant of the RGB input color space. Hence the output color values are effectively divided into four quadrants each designated by a two bit tag. Consequently the output color values for each tag value for a particular line are highly spaced apart in the output color space, enabling anti-aligning of memory patterns.

Where the two bit tags do not match a cache miss is recorded by the data cache controller and the corresponding required memory read is initiated by the data cache controller with the cache look up process being stalled until all values for that line corresponding to that two bit tag entry are read from an external memory and stored in the cache. This involves the reading of the relevant line of the color conversion table stored in the external memory. The process 575 of Fig. 63 is carried out for each of the memory banks eg. 573 of Fig. 62 resulting, depending on the cache contents, in a time interval elapsing before the results eg. 586 are output from each corresponding memory bank. Each of the eight 32 bit sets of data 586 are then forwarded to the main data path unit (242) which carries out the aforementioned interpolation process (Fig. 62) in three stages 590 - 592 to each of the colored channels simultaneously and in a pipelined manner so as to produce four color outputs 595 for sending to a printer device.

Experiments have shown that the caching mechanism as described with reference to Figs. 62 and 63 can be advantageously utilized as typical images have a cache miss-rate on average requiring between 0.01 and 0.03 cache line fetches per pixel. The utilization of the caching mechanism therefore leads to substantially reduced requirements, in the typical case, for memory accesses outside of the data cache.

The instruction encoding for both color space conversion modes (Fig. 10) utilized by the co-processor has the following structure:

Table 12A: Instruction Encoding for Color Space Conversion

Operand	Description	Internal Format	External Format
Operand A	source pixels	pixels	packed stream
Operand B	multi output channel color conversion tables	other	multi channel csc tables
Operand C	Interval and Fraction Tables	-	I&F table format
Result	pixels	pixels	packed stream
	bytes	unpacked bytes	unpacked bytes, packed stream

The instruction field encoding for color space conversion instruction is illustrated in Fig. 64 with the following minor opcode encoding for the color
 5 conversion instructions.

Table 13: Minor Opcode Encoding for Color Conversion Instructions

Field	Description
trans[3:0]	0 = do not apply translation and clamping step to corresponding output value on this channel
M	0 = single channel color table format 1 = multi channel color table format

10 Fig. 65 shows a method of converting a stream of RGB pixels into CYMK color values according to the MOGCS mode. In step S₁, a stream of 24 bit RGB pixels are received by the pixel organiser 246 (Fig. 2). In step S₂, the pixel organiser 246 determines the 4 bit interval values and the 8 bit fractional values of each input pixel from lookup tables, in the manner
 15 previously discussed with respect to Figs. 56 and 57. The interval and fractional values of the input pixel designate which intervals and fractions

along the intervals in which the input pixel lies. In step S_3 , the main data path unit 242 concatenates the 4 bit intervals of the red, green and blue color components of the input pixel to form a 12 bit address word and supplies this 12 bit address word to the data cache controller 240 (Fig. 2). In step S_4 , the data cache controller 240 converts this 12 bit address word into 8 different 9 bit addresses, in the manner previously discussed with respect to Table 12 and Fig. 62. These 8 different addresses designate the location of the 8 output color values $CV(P0)$ - $CV(P7)$ in the respective memory banks 573 (Fig. 62) of the data cache 230 (Fig. 2). In step S_5 , the data cache controller 240 (Fig. 2) remaps the 8 different 9 bit addresses in the manner described previously with respect to Fig. 63. In this way, the most significant bit of the red and green 4 bit intervals are mapped to the two most significant bits of the 9 bit addresses.

In step S_6 , the data cache controller 240 then compares the two most significant bits of the 9 bit addresses with respective 2 bit tags in memory 587 (Fig. 63). If the 2 bit tag does not correspond to the two most significant bits of the 9 bit addresses, then the output color values $CV(P0)$ - $CV(P7)$ do not exist in the cache memory 230. Hence, in step S_7 , all the output color values corresponding to the 2 bit tag entry for that line are read from external memory into the data cache 230. If the 2 bit tag corresponds to these two most significant bits of the 9 bit addresses, then the data cache controller 240 retrieves in step S_8 the eight output color values $CV(P0)$ - $CV(P7)$ in the manner discussed previously with respect to Fig. 62. In this way, the eight output color values $CV(P0)$ - $CV(P7)$ surrounding the input pixel are retrieved by the main data path unit 242 from the data cache 230. In step S_9 , the main data path unit 242 interpolates the output color values $CV(P0)$ - $CV(P7)$ utilising the fractional values determined in step S_2 and outputs the interpolated output color values.

It will be evident to the man skilled in the art, that the storage space of the data cache storage may be reduced further by dividing the RGB color space and the corresponding output color values into more than four quadrants, for example 32 blocks. In the latter case, the data cache can have
 5 the capacity of storing only a 1/32 block of output color values.

It will also be evident to the man skilled in the art, that the data caching arrangement utilized in the MOGCS mode can also be used in a single output general conversion mode. Hence, in the latter mode the storage space of the data cache can also be reduced.

10 3.17.3 JPEG Coding/Decoding

It is well known that a large number of advantages can be obtained from storing images in a compressed format especially in relation to the saving of memory and the speed of transferring images from one place to another. Various popular standards have arisen for image compression.
 15 One very popular standard is the JPEG standard and for a full discussion of the implementation of this standard reference is made to the well known text *JPEG: Still Image Data Compression Standard* by Pennebaker and Mitchell published 1993 by Van Nostrand Reinhold. The co-processor 224 utilizes a subset of the JPEG standard in the storage of images. The JPEG standard
 20 has the advantage that large factor compression can be gained with the retention of substantial image quality. Of course, other standards for storing compressed images could be utilized. The JPEG standard is well-known to those skilled in the art, and the various JPEG alternative implementations readily available in the marketplace from manufacturers including JPEG core
 25 products for incorporation into ASICS.

The co-processor 224 implements JPEG compression and decompression of images consisting of 1, 3 or 4 color components. One-color-component images may be meshed or unmeshed. That is, a single-color-component can be extracted from meshed data or extracted from unmeshed
 30 data. An example of meshed data is three-color components per pixel datum

(i.e., RGB per pixel datum), and an example of unmeshed data is where each color component for an image is stored separately such that each color component can be processed separately. For three color component images the co-processor 224 utilizes one pixel per word, assuming the three color channels to be encoded in the lowest three bytes.

The JPEG standard decomposes an image into small two dimensional units called minimum coded units (MCU). Each minimal coded unit is processed separately. The JPEG coder 241 (Fig. 2) is able to deal with MCU's which are 16 pixels wide and 8 pixels high for down sampled images or
 10 MCU's which are 8 pixels wide and 8 pixels high for images that are not to be down sampled.

Turning now to Fig. 66, there is illustrated the method utilized for down sampling three component images.

The original pixel data 600 is stored in the MUV buffer 250 (Fig. 2) in
 15 a pixel form wherein each pixel 601 comprises Y, U and V components of the YUV color space. This data is first converted into a MCU unit which comprises four data blocks 601 - 604. The data blocks comprise the various color components, with the Y component being directly sampled 601, 602 and the U and V components being sub-sampled in the particular example of Fig.
 20 13 to form blocks 603, 604. Two forms of sub-sampling are implemented by the co-processor 224, including direct sampling where no filtering is applied and odd pixel data is retained while even pixel data is discarded. Alternatively, filtering of the U and V components can occur with averaging of adjacent values taking place.

25 An alternative form of JPEG sub-sampling is four color channel sub-sampling as illustrated in Fig. 67. In this form of sub-sampling, pixel data blocks of 16 x 8 pixels 610 each have four components 611 including an opacity component (O) in addition to the usual Y, U, V components. This pixel data 410 is sub-sampled in a similar manner to that depicted in Fig. 66

However, in this case, the opacity channel is utilized to form data blocks 612, 613.

Turning now to Fig. 68, there is illustrated the JPEG coder 241 of Fig. 2 in more detail. The JPEG encoder/decoder 241 is utilized for both JPEG encoding and decoding. The encoding process receives block data via bus 620 from the pixel organizer 246 (Fig. 2). The block data is stored within the MUV buffer 250 which is utilized as a block staging area. The JPEG encoding process is broken down into a number of well defined stages. These stages include:

1. taking a discrete cosine transform (DCT) via DCT unit 621;
2. quantising the DCT output 622;
3. placing the quantized DCT co-efficients in a zig zag order, also carried out by quantizer unit 622;
4. predictively encoding the DC DCT co-efficients and run length encoding the AC DCT co-efficients carried out by co-efficient coder 623; and
5. variable length encoding the output of the co-efficients coder stage, carried out by Huffman coder unit 624. The output is fed via multiplexer 625 and Rbus 626 to the result organizer 629 (Fig. 2).

The JPEG decoding process is the inverse of JPEG encoding with the order of operations reversed. Hence, the JPEG decoding process comprizes the steps of inputting on Bus 620 a JPEG block of compressed data. The compressed data is transferred via Bus 630 to the Huffman coder unit 624 which Huffman decodes data into DC differences and AC run lengths. Next, the data is forwarded to the co-efficients coder 623 which decodes the AC and DC co-efficients and puts them into their natural order. Next, the quantizer unit 622 dequantizes the DC co-efficients by multiplying them by a corresponding quantization value. Finally, the DCT unit 621 applies an inverse discrete cosine transform to restore the original data which is then transferred via Bus 631 to the multiplexer 625 for output via Bus 626 to the Result Organizer. The JPEG coder 241 operates in the usual manner via

standard CBus interface 632 which contains the registers set by the instructions controller in order to begin operation of the JPEG coder. Further, both the quantizer unit 622 and the Huffman coder 624 require certain tables which are loaded in the data cache 230 as required. The table data is
 5 accessed via an OBus interface unit 634 which connects to the operand organizer B unit 247 (Fig. 2) which in turn interacts with the data cache controller 240.

The DCT unit 621 implements forward and inverse discrete cosine transforms on pixel data. Although many different types of DCT
 10 transforming implementations are known and discussed in the *Still Image Data Compression Standard* (ibid), the DCT 621 implements a high speed form of transform more fully discussed in the section herein entitled *A Fast DCT Apparatus*, which may implement a DCT transform operation in accordance with the article entitled *A Fast DCT-SQ Scheme for Images* by
 15 Arai et. al., published in The Transactions of the IEICE, Vol E71, No. 11, November 1988 at page 1095.

The quantizer 622 implements quantization and dequantization of DCT components and operates via fetching relevant values from corresponding tables stored in the data cache via the OBus interface unit 634.
 20 During quantization, the incoming data stream is divided by values read from quantization tables stored in the data cache. The division is implemented as a fixed point multiply. During dequantization, the data stream is multiplied by values kept in the dequantization table.

Turning to Fig. 69, there is illustrated the dequantizer 622 in more
 25 detail. The quantizer 622 includes a DCT interface 640 responsible for passing data to and receiving data from the DCT module 621 via a local Bus. During quantization, the quantizer 622 receives two DCT co-efficients per clock cycle. These values are written to one of the quantizers internal buffers 641, 642. The buffers 641, 642 are dual ported buffers used to buffer
 30 incoming data. During quantization, co-efficient data from the DCT sub-

module 621 is placed into one of the buffers 641, 642. Once the buffer is full, the data is read from the buffer in a zig zag order and multiplied by multiplier 643 with the quantization values received via OBus interface unit 634. The output is forwarded to the co-efficient coder 623 (Fig. 68) via co-efficient coder interface 645. While this is happening, the next block of co-efficients is being written to the other buffer. During JPEG decompression, the quantizer module dequantizes decoded DCT co-efficients by multiplying them by values stored in the table. As the quantization and dequantization operations are mutually exclusive, the multiplier 643 is utilized during quantization and dequantization. The position of the co-efficient within the block of 8x8 values is used as the index into the dequantization table.

As with quantization, the two buffers 641, 642 are utilized to buffer incoming co-efficient data from the co-efficient coder 623 (Fig. 68). The data is multiplied with its quantization value and written into the buffers in reverse zig zag order. Once full, the dequantized co-efficients are read out of the utilized buffer in natural order, two at a time, and passed via DCT interface 640 to the DCT sub-module 621 (Fig. 68). Hence the co-efficients coder interface module 645 is responsible for interfacing to the co-efficients coder and passes data and receives data from the coder via a local Bus. This module also reads data from buffers in zig zag order during compression and writes data to the buffers in reverse zig zag order during decompression. Both the DCT interface module 640 and the CC interface module 645 are able to read and write from buffers 641, 642. Hence, address and control multiplexer 647 is provided to select which buffer each of these interfaces is interacting with under the control of a control module 648, which comprizes a state machine for controlling all the various modules in the quantizer. The multiplier 643 can be a 16x8, 2's complement multiplier which multiplies DCT co-efficients by quantization table values.

Turning again to Fig. 68, the co-efficient coder 623 performs the functions of:

(a) predictive encoding/decoding of DC co-efficients in JPEG mode;
and

(b) run length encoding/decoding of AC co-efficients in JPEG mode.

Preferably, the co-efficient coder 623 is also able to be utilized for
5 predictive encoding/decoding of pixels and memory copy operations as required
independently of JPEG mode operation. The co-efficient coder 623
implements predictive and run length encoding and decoding of DC and AC co-
efficients as specified in the Pink Book. A standard implementation of
predictive encoding and predictive decoding in addition to JPEG AC co-
10 efficients run length encoding and decoding as specified in the JPEG
standard is implemented.

The Huffman coder 624 is responsible for Huffman encoding and
decoding of the JPEG data train. In Huffman encoding mode, the run length
encoded data is received from the co-efficients coder 623 and utilized to
15 produce a Huffman stream of packed bytes. Alternatively, or in addition, in
Huffman decoding, the Huffman stream is read from the PBus interface 620
in the form of packed bytes and the Huffman decoded co-efficients are
presented to the co-efficient coder module 623. The Huffman coder 624
utilizes Huffman tables stored in the data cache and accessed via OBus
20 interface 634. Alternatively, the Huffman table can be hardwired for
maximum speed.

When utilising the data cache for Huffman coding, the eight banks of
the data cache store data tables as follows with the various tables being
described in further detail hereinafter.

Table 14: Huffman and Quantization Tables as stored in Data Cache

Bank	Description
0	This bank hold the 256, 16 bit entries of a EHUFDCO_DC_1 or EHUFDCO table. The least significant bit of the index chooses between the two 16 bit items in the 32 bit word. All 128 lines of this bank of memory are used.
1	This bank holds the 256, 16 bit entries of a EHUFDCO_DC_2 table. The least significant bit of the index chooses between the two 16 bit items in the 32 bit word. All 128 lines of this bank of memory are used.
2	This bank holds the 256, 16 bit entries of a EHUFDCO_AC_1 table. The least significant bit of the index chooses between the two 16 bit items in the 32 bit word. All 128 lines of this bank of memory are used.
3	This bank holds the 256, 16 bit entries of a EHUFDCO_AC_2 table. The least significant bit of the index chooses between the two 16 bit items in the 32 bit word. All 128 lines of this bank of memory are used.
4	This bank holds the 256, 4 bit entries of a EHUFDCI_DC_1 or EHUFDCI table, as well as the 256, 4 bit entries of a EHUFDCI_DC_2 table. All 128 lines of this bank of memory are used.
5	This bank holds the 256, 4 bit entries of a EHUFDCI_AC_1 table, as well as the 256, 4 bit entries of a EHUFDCI_AC_2 table. All 128 lines of this bank of memory are used.
6	Not used
7	This bank holds the 128, 24 bit entries of the quantization table. It occupies the least significant 3 bytes of all 128 lines of this bank of memory.

Turning now to Fig. 70, the Huffman coder 624 consists primarily of two independent blocks being an encoder 660 and a decoder 661. Both blocks 660,661 share the same OBus interface via a multiplexer module 662. Each block has its own input and output with only one block active at a time, depending on the function performed by the JPEG encoder.

a. Encoding

During encoding in JPEG mode, Huffman tables are used to assign codes of varying lengths (up to 16 bits per code) to the DC difference values and to the AC run-length values, which are passed to the HC submodule from the CC submodule. These tables have to be preloaded into the data cache before the start of the operation. The variable length code words are then concatenated with the additional bits for DC and AC co-efficients (also passed from the CC submodule, then packed into bytes. A X'00 byte is stuffed in if an X'FF byte is obtained as a result of packing. If there is a need for an RST_m marker it is inserted. This may require byte padding with "1" bits of the last Huffman code and X'00 byte stuffing if the padded byte results in X'FF. The need for an RST_m marker is signalled by the CC submodule. The HC submodule inserts the EOI marker at the end of image, signalled by the "final" signal on the PBus-CC slave interface. The insertion procedure of the EOI marker requires similar packing, padding and stuffing operations as for RST_m markers. The output stream is finally passed as packed bytes to the Result Organizer 249 for writing to external memory.

In non-JPEG mode data is passed to the encoder from the CC submodule (PBus-CC slave interface) as unpacked bytes. Each byte is separately encoded using tables preloaded into the cache (similarly to JPEG mode), the variable length symbols are then assembled back into packed bytes and passed to the Results Organizer 249. The very last byte in the output stream is padded with 1's.

b. Decoding

Two decoding algorithms are implemented: fast (real time) and slow (versatile). The fast algorithm works only in JPEG mode, the versatile one works both in JPEG and non-JPEG modes.

The fast JPEG Huffman decoding algorithm maps Huffman symbols to either DC difference values or AC run-length values. It is specifically tuned for JPEG and assumes that the example Huffman tables (K3, K4, K5

and K6) were used during compression. The same tables are hard wired in to the algorithm allowing decompression without references to the cache memory. This decoding style is intended to be used when decompressing images to be printed where certain data rates need to be guaranteed. The data rate for the HC submodule decompressing a band (a block between RST_m markers) is almost one DC/AC co-efficient per clock cycle. One clock cycle delay between the HC submodule and CC sub-module may happen for each X'00 stuff byte being removed from the data stream, however this is strongly data dependent.

The Huffman decoder operates in a faster mode for the extraction of one Huffman symbol per clock cycle. The fast Huffman decoder is described in the section herein entitled *Decoder of Variable Length Codes*.

Additionally, the Huffman decoder 661 also implements a heap-based slow decoding algorithm and has a structure 670 as illustrated in Fig. 71.

For a JPEG encoded stream, the STRIPPER 671 removes the X'00 stuff bytes, the X'FF fill bytes and RST_m markers, passing Huffman symbols with concatenated additional bits to the SHIFTER 672. This stage is bypassed for Huffman-only coded streams.

The first step in decoding a Huffman symbol is to look up the 256 entries HUFVAL table stored in the cache addressing it with the first 8 bits of the Huffman data stream. If this yields a value (and the true length of the corresponding Huffman symbol), the value is passed on to the OUTPUT FORMATTER 676, and the length of the symbol and the number of the additional bits for the decoded value are fed back to the SHIFTER 672 enabling it to pass the relevant additional bits to the OUTPUT FORMATTER 676 and align the new front of the Huffman stream presented to the decoding unit 673. The number of the additional bits is a function of the decoded value. If the first look up does not result in a decoded value, which means that the Huffman symbol is longer than 8 bits, the heap address is calculated and successive heap (located in the cache, too) accesses are performed following the algorithm until a match is found or an "illegal Huffman symbol"

condition met. A match results in identical behavior as in case of the first match and "illegal Huffman symbol" generates an interrupt condition.

The algorithm for heap-based decoding algorithm is as follows:

```

loop until end of image
5      set symbol length N to 8
      get first 8 bits of the input stream into INDEX
      fetch HUFVAL(INDEX)
      if HUFVAL(INDEX) == 00xx 0000 111 -- (ILL)
          signal "illegal Huffman symbol"
10      exit
      elseif HUFVAL(INDEX) == 1nnn eeee eeee -- (HIT)
          pass nnn bits to eeee eeee as the value
          pass symbol length N = decimal (nnn)/*000
          as symbol length 8*/
15      adjust the input stream
          break
      else/* HUFVAL (INDEX) == 01iii iii iii -- (MISS)*/
          set HEAPINDEX = ii iii iii -- (we assume heapbase = 0)
          set N = 9
20      if 9th bit of the input stream == 0
          increment HEAPINDEX
          fi
      fetch VALUE = HEAP (HEAPINDEX) -- (code for 9th bit)
      loop
25      if VALUE == 0001 0000 1111 -- (ILL)
          signal "illegal Huffman symbol"
          exit
          elseif VALUE == 1000 eeee eeee
              pass eeee eeee as the value
30      pass symbol length N

```

```

                                adjust the input stream
                                break
                                else/* VALUE == 01iii iii iii -- (MISS) */
                                    set N = N + 1 -- (HEAPINDEX = ii iii iii)
5                                if Nth bit of the input stream == 0
                                    increment HEAPINDEX
                                fi
                                fetch VALUE = HEAP (HEAPINDEX)
                                pool
10                                pool

```

The STRIPPER 671 removes any X00 stuff bytes, XFF fill bytes and RST_m markers from the incoming JPEG 671 coded stream and passes "clean" Huffman symbols with concatenated additional bits to the shifter 672.

15 There are no additional bits in Huffman-only encoding, so in this mode the passed stream consists of Huffman symbols only.

The shifter 672 block has a 16 bit output register in which it presents the next Huffman symbol to the decoding unit 673 (bitstream running from MSB to LSB). Often the symbol is shorter than 16 bits, but it is up to the

20 decoding unit 673 to decide how many bits are currently being analysed. The shifter 672 receives a feedback 678 from the decoding unit 673, namely the length of the current symbol and the length of the following additional bits for the current symbol (in JPEG mode), which allows for a shift and proper alignment of the beginning of the next symbol in the shifter 672.

25 The decoding unit 673 implements the core of the heap based algorithm and interfaces to the data cache via the OBus 674. It incorporates a Data Cache fetch block, lookup value comparator, symbol length counter, heap index adder and a decoder of the number of the additional bits (the decoding is based on the decoded value). The fetch address is interpreted as

30 follows:

Table 15: Fetch Address

Field (bits)	Description
[32:25]	Index into dequantization tables.
[24:19]	Not used.
[18:9]	Index into the heap.
[8:0]	Index into Huffman decode table.

The OUTPUT FORMATTER block 676 packs decoded 8-bit values
 5 (standalone Huffman mode), or packs 24-bit value + additional bits + RST_m
 marker information (JPEG mode) into 32-bit words. The additional bits are
 passed to the OUTPUT FORMATTER 676 by the shifter 672 after the
 decoding unit 673 decides on the start position of the additional bits for the
 current symbol. The OUTPUT FORMATTER 673 also implements a 2 deep
 10 FIFO buffer using a one word delay for prediction of the final value word.
 During the decoding process, it may happen that the shifter 672 (either fast or
 slow) tries to decode the trailing padding bits at the end of the input
 bitstream. This situation is normally detected by the shifter and instead of
 asserting the "illegal symbol" interrupt, it asserts a "force final" signal.
 15 Active "force final" signal forces the OUTPUT FORMATTER 676 to signal the
 last but one decoded word as "final" (this word is still present in the FIFO)
 and discard the very last word which does not belong to the decoded stream.

The Huffman encoder 660 of Fig. 70 is illustrated in Fig. 72 in more
 detail. The Huffman encoder 660 maps byte data into Huffman symbols via
 20 look up tables and includes a encoding unit 681, a shifter 682 and a OUTPUT
 FORMATTER 683 with the lookup tables being accessed from the cache.

Each submitted value 685 is coded by the encoding unit 681 using
 coding tables stored in the data cache. One access to the cache 230 is needed
 to encode a symbol, although each value being encoded requires two tables,
 25 one that contains the corresponding code and the other that contains the code
 length. During JPEG compression, a separate set of tables is needed for AC

and DC co-efficients. If subsampling is performed, separate tables are required for subsampled and non subsampled components. For non-JPEG compression, only two tables (code and size) are needed. The code is then handled by the shifter 682 which assembles the outgoing stream on bit level.

5 The Shifter 682 also performs RST_m and EOI markers insertion which implies byte padding, if necessary. Bytes of data are then passed to the OUTPUT FORMATTER 683 which does stuffing (with X'00 bytes), filling with X'FF bytes, also the FF bytes leading the marker codes and formatting to packed bytes. In the non-JPEG mode, only formatting of packed bytes is

10 required.

Insertion of X'FF bytes is handled by the shifter 682, which means that the output formatter 683 needs to tell which bytes passed from the shifter 682 represent markers, in order to insert an X'FF byte before. This is done by having a register of tags which correspond to bytes in the shifter 682.

15 Each marker, which must be on byte boundaries anyway, is tagged by the shifter 682 during marker insertion. The packer 683 does not insert stuff bytes after the X'FF bytes preceding the markers. The tags are shifted synchronously with the main shift register.

The Huffman encoder uses four or eight tables during JPEG

20 compression, and two tables for straight Huffman encoding. The tables utilized are as follows:

Table 16: Tables Used by the Huffman Encoder

Name	Size	Description
EHUFSI	256	Huffman code sizes. Used during straight Huffman encoding. Uses the coded value as an index.
EHUFCO	256	Huffman code values used during straight Huffman encoding. Uses the coded value as an index.
EHUFSI_DC_1	16	Huffman codes sizes used to code DC co-efficients during JPEG compression. Uses magnitude category as the index.
EHUFCO_DC_1	16	Huffman code values used to code DC co-efficients during JPEG compression. Uses magnitude category as an index. Used for subsampled blocks.
EHUFSI_DC_2	16	Huffman code sizes used to code DC co-efficients during JPEG compression. Uses magnitude category as an index. Used for subsampled blocks.
EHUFCO_DC_2	16	Huffman code sizes used to code DC co-efficients during JPEG compression. Uses magnitude category as an index. Used for subsampled blocks.

Name	Size	Description
EHUFSI_AC_1	256	Huffman code sizes used to code AC coefficients during JPEG compression. Uses magnitude category and run-length as an index.
EHUFCO_AC_1	256	Huffman code sizes used to code AC coefficients during JPEG compression. Uses magnitude category and run-length as an index.
EHUFSI_AC_2	256	Huffman code sizes used to code AC coefficients during JPEG compression for subsampled components. Uses magnitude category and run-length as an index.
EHUFCO_AC_2	256	Huffman code sizes used to code AC coefficients during JPEG compression for subsampled components. Uses magnitude category and run-length as an index.

3.17.4 Table Indexing

Huffman tables are stored locally by the co-processor data cache 230.

- 5 The data cache 230 is organized as a 128 line, direct mapped cache, where each line comprises 8 words. Each of the words in a cache line are separately addressable, and the Huffman decoder uses this feature to simultaneously access multiple tables. Because the tables are small (≤ 256 entries), the 32 bit address field of the OBus can carry indexes into multiple tables.

- 10 As noted previously, in JPEG slow decoding mode, the data cache is utilized for storing various Huffman tables. The format of the data cache is as follows:

Table 17: Bank Address for Huffman and Quantization Tables

Bank	Description
0 to 3	These banks hold the 1024, 16 bit entries of the heap. The least significant index bit selects between the two 16 bit words in each bank. All 128 lines of the four banks of memory are used.

Bank	Description
4	This bank holds the 512, least significant 8 bits of the 12 bit entries of the DC Huffman decode table. The least significant two bits of the index chooses between the four, byte items in the 32 bit word. All 128 line of this bank of memory are used.
5	This bank holds the 512, least significant 8 bits of the 12 bit entires of the AC Huffman decode table. The least significant two bits of the index chooses between the four, byte items in the 32 bit word. All 128 lines of this bank of memory are used.
6	This bank holds the most significant 4 bits of both the DC and AC Huffman decode tables. The least significant 2 bits of each index chooses between the 4 respective nibbles within each word.
7	This bank holds the 128, 24 bit entires of the quantization table. It occupies the least significant 3 bytes of all 128 lines of this bank of memory.

Prior to each JPEG instruction being executed by the JPEG coder 241 (Fig. 2) the appropriate image width value in the image dimensions register (PO_IDR) or (RO_IDR) must be set. As with other instructions, the length of the instruction refers to the number of input data items to be processed. This includes any padding data and accounts for any sub-sampling options utilized and for the number of color channels used.

All instructions issued by the co-processor 224 may utilize two facilities for limiting the amount of output data produced. These facilities are most useful for instructions where the input and output data sizes are not the same and in particular where the output data size is unknown, such as for JPEG coding and decoding. The facilities determine whether the output data

is written out or merely discarded with everything else being as if the instruction was properly processed. By default, these facilities are normally disabled and can be enabled by enabling the appropriate bits in the RO_CFG register. JPEG instructions however, include specific option for setting these bits. Preferably, when utilising JPEG compression, the co-processor 224 provides facilities for "cutting" and "limiting" of output data.

Turning to Fig. 73, there is now described the process of cutting and limiting. An input image 690 may be of a certain height 691 and a certain width 692. Often, only a portion of the image is of interest with other portions being irrelevant for the purposes of printing out. However, the JPEG encoding system deals with 8x8 blocks of pixels. It may be the case that, firstly, the image width is not an exact multiple of 8 and additionally, the section of interest comprising MCU 695 does not fit across exact boundaries. An output cut register, RO_cut specifies the number of output bytes at 696 at the beginning of the output data stream to discard. Further, an output limit register, RO_LMT specifies the maximum number of output bytes to be produced. This count includes any bytes that do not get written to memory as a result of the cut register. Hence, it is possible to target a final output byte 698 beyond which no data is to be outputted.

There are two particular cases where the cut and limited functionality of the JPEG decoder is considered to be extremely useful. The first case, as illustrated in Fig. 74, is the extraction or decompression of a sub-section 700 of one strip 701 of a decompressed image. The second useful case is illustrated in Fig. 75 wherein the extraction or decompression of a number of complete strips (eg. 711, 712 and 713) is required from an overall image 714.

The instruction format and field encoding for JPEG instructions is as illustrated in Fig. 76. The minor opcode fields are interpreted as follows:

Table 18: Instruction Word - Minor Opcode Fields

Field	Description
D	0 = encode(compress) 1 = decode(decompress)
M	0 = single color channel 1 = multi channel
4	0 = three channel 1 = four channel
S	0 = do not use a sub/up sampling regime 1 = use a subsampling regime
H	0 = use fast Huffman coding 1 = use general purpose Huffman coding
C	0 = do not use cut register 1 = use cut register
T	0 = do not truncate on output 1 = truncate on output
F	0 = do not low pass filter before subsampling 1 = low pass filter before subsampling

3.17.5 Data Coding Instructions

Preferably, the co-processor 224 provides for the ability to utilize portions of the JPEG coder 241 of Fig. 2 in other ways. For example, Huffman coding is utilized for both JPEG and many other methods of compression. Preferably, there is provided data coding instructions for manipulating the Huffman coding unit only for hierarchial image decompression. Further, the run length coder and decoder and the predictive coder can also be separately utilized with similar instructions.

3.17.6 A Fast DCT Apparatus

Conventionally, a discrete cosine transform (DCT) apparatus as shown in Fig. 77 performs a full two-dimensional (2-D) transformation of a block of 8×8 pixels by first performing a 1-D DCT on the rows of the 8×8 pixel block. It then performs another 1-D DCT on the columns of the 8×8 pixel block. Such an apparatus typically consists of an input circuit 1096, an arithmetic circuit 1104, a control circuit 1098, a transpose memory circuit 1090, and an output circuit 1092.

The input circuit 1096 accepts 8-bit pixels from the 8x8 block. The input circuit 1096 is coupled by intermediate multiplexers 1100, 1102 to the arithmetic circuit 1004. The arithmetic circuit 1104 performs mathematical operations on either a complete row or column of the 8x8 block. The control circuit 1098 controls all the other circuits, and thus implements the DCT algorithm. The output of the arithmetic circuit is coupled to the transpose memory 1090, register 1095 and output circuit 1092. The transpose memory is in turn connected to multiplexer 1100, which provides output to the next multiplexer 1102. The multiplexer 1102 also receives input from the register 1094. The transpose circuit 1090 accepts 8x8 block data in rows and produces that data in columns. The output circuit 1092 provides the coefficients of the DCT performed on a 8x8 block of pixel data.

In a typical DCT apparatus, it is the speed of the arithmetic circuit 1104 that basically determines the overall speed of the apparatus, since the arithmetic circuit 1104 is the most complex.

The arithmetic circuit 1104 of Fig. 77 is typically implemented by breaking the arithmetic process down into several stages as described hereinafter with reference to Fig. 78. A single circuit is then built that implements each of these stages 1114, 1148, 1152, 1156 using a pool of common resources, such as adders and multipliers. Such a circuit 1104 is mainly disadvantageous due to it being slower than optimal, because a single, common circuit is used to implement the various stages of circuit 1104. This includes a storage means used to store intermediate results. Since the time allocated for the clock cycle of such a circuit must be greater or equal to the time of the slowest stage of the circuit, the overall time is potentially longer than the sum of all the stages.

Fig. 78 depicts a typical arithmetic data path, in accordance with the apparatus of Fig. 77, as part of a DCT with four stages. The drawing does not reflect the actual implementation, but instead reflects the functionality. Each of the four stages 1144, 1148, 1152, and 1156 is implemented using a

single, reconfigurable circuit. It is reconfigured on a cycle-by-cycle basis to implement each of the four arithmetic stages 1144, 1148, 1152, and 1156 of the 1-D DCT. In this circuit, each of the four stages 1144, 1148, 1152, and 1156 uses pool of common resources (e.g. adders and multipliers) and thus
 5 minimises hardware.

However, the disadvantage of this circuit is that it is slower than optimal. The four stages 1144, 1148, 1152, and 1156 are each implemented from the same pool of adders and multipliers. The period of the clock is therefore determined by the speed of the slowest stage, which in
 10 this example is 20 ns (for block 1144). Adding in the delay (2ns each) of the input and output multiplexers 1146 and 1154 and the delay (3ns) of the flip-flop 1150, the total time is 27 ns. Thus, the fastest this DCT implementation can run at is 27 ns.

Pipelined DCT implementations are also well known. The drawback
 15 with such implementations is that they require large amounts of hardware to implement. Whilst the present invention does not offer the same performance in terms of throughput, it offers an extremely good performance/size compromise, and good speed advantages over most of the current DCT implementations.

Fig. 79 shows a block diagram of the preferred form of discrete cosine transform unit utilized in the JPEG coder 241 (Fig. 2) where pixel data is inputted to an input circuit 1126 which captures an entire row of 8-bit pixel data. The transpose memory 1118 converts row formatted data into column formatted data for the second pass of the two dimensional discrete cosine
 25 transform algorithm. Data from the input circuit 1126 and the transpose memory 1118 is multiplexed by multiplexer 1124, with the output data from multiplexer 1124 presented to the arithmetic circuit 1122. Results data from the arithmetic circuit 1122 is presented to the output circuit 1120 after the second pass of the process. The control circuit 1116 controls the flow of
 30 data through the discrete cosine transform apparatus.

During the first pass of the discrete cosine transform process row data from the image to be transformed, or transformed image coefficients to be transformed back to pixel data is presented to the input circuit 1126. During this first pass, the multiplexer 1124 is configured by the control circuit 1116
 5 to pass data from the input circuit 1126 to the arithmetic circuit 1122.

Turning to Fig. 80, there is shown the structure of the arithmetic circuit 1122 in more detail. In the case of performing a forward discrete cosine transform, the results from the forward circuit 1138 which is utilized to calculate the forward discrete cosine transform is selected via the multiplexer
 10 1142, which is configured in this way by the control circuit 1116. When an inverse discrete cosine transform is to be performed, the output from the inverse circuit 1140 is selected via the multiplexer 1142, as controlled by the control circuit 1126. During the first pass, after each row vector has been processed by the arithmetic circuit 1122 (configured in the appropriate way by
 15 control circuit 1116), that vector is written into the transpose memory 1118. Once all eight row vectors in an 8x8 block have been processed and written into the transpose memory 1118, the second pass of the discrete cosine transform begins.

During the second pass of either the forward or inverse discrete cosine
 20 transforms, column ordered vectors are read from the transpose memory 1118 and presented to the arithmetic circuit 1122 via the multiplexer 1124. During this second pass, the multiplexer 1124 is configured by the control circuit to ignore data from the input circuit 1136 and pass column vector data from the transpose memory 1118 to the arithmetic circuit 1122. The
 25 multiplexer 1142 in the arithmetic circuit 1122 is configured by the control circuit 1116 to pass results data from the inverse circuit 1140 to the output of the arithmetic circuit 1122. When results from the arithmetic circuit 1122 are available, they are captured by the output circuit 1120 under direction from the control circuit 1116 to be outputted sometime later.

The arithmetic circuit 1122 is completely combinatorial, in that is there are no storage elements in the circuit storing intermediate results. The control circuit 1116 knows how long it takes for data to flow from the input circuit 1136, through the multiplexer 1124 and through the arithmetic circuit 1122, and so knows exactly when to capture the results vector from the outputs of the arithmetic circuit 1122 into the output circuit 1120. The advantage of having no intermediate stages in the arithmetic circuit 1122 is that no time is wasted getting data in and out of intermediate storage elements, but also the total time taken for data to flow through the arithmetic circuit 1122 is equal to the sum of all the internal stages and not N times the delay of the longest stage (as with conventional discrete cosine transform implementations), where N is the number of stages in the arithmetic circuit.

Referring to Fig. 81, the total time delay is simply the sum of the four stage 1158, 1160, 1162, 1164, which is $20\text{ ns} + 10\text{ ns} + 12\text{ ns} + 15\text{ ns} = 57\text{ ns}$, which is faster than the circuit depicted in Fig. 78. The advantage of this circuit is that it provides an opportunity to reduce the overall system's clock period. Assuming that four clock cycles are allocated to getting a result from the circuit depicted in Fig. 81, the fastest run time for the entire DCT system would be $57/4\text{ ns}$ (14.25 ns), which is a significant improvement over the circuit in Fig. 78 which only allows for a DCT clock period of substantially 27 ns .

An exemplary implementation of the present DCT apparatus might, but not necessarily, use the DCT algorithm proposed in the paper to The Transactions of the IEICE, Vol. E 71, No. 11, November 1988, entitled *A Fast DCT-SQ Scheme for Images* at page 1095 by Yukihiro Arai, Takeshi Agui and Masayuki Nakajima. By implementing this algorithm in hardware, it can then easily be placed in the current DCT apparatus in the arithmetic circuit 1122. Likewise, other DCT algorithms may be implemented in hardware in place of arithmetic circuit 1122.

3.17.7 Huffman Decoder

The aspects of the following embodiment relate to a method and apparatus for variable-length codes interleaved with variable length bit fields. In particular, the embodiments of the invention provide efficient and fast, single stage (clock cycle) decoding of variable-length coded data in which byte aligned and not variable length encoded data is removed from the encoded data stream in a separate pre-processing block. Further, information about positions of the removed byte-aligned data is passed to the output of the decoder in a way which is synchronous with the data being decoded. In addition, it provides fast detection and removal of not byte-aligned and not variable length encoded bit fields that are still present in the pre-processed input data.

The preferred embodiment of the present invention preferably provides for a fast Huffman decoder capable of decoding a JPEG encoded data at a rate of one Huffman symbol per clock cycle between marker codes. This is accomplished by means of separation and removal of byte aligned and not Huffman encoded marker headers, marker codes and stuff bytes from the input data first in a separate pre-processing block. After the byte aligned data is removed, the input data is passed to a combinatorial data-shifting block, which provides continuous and contiguous filling up of the data decode register that consequently presents data to a decoding unit. Positions of markers removed from the original input data stream are passed on to a marker shifting block, which provides shifting of marker position bits synchronously with the input data being shifted in the data shifting block.

The decoding unit provides combinatorial decoding of the encoded bit field presented to its input by the data decode register. The bit field is of a fixed length of n bits. The output of the decoding unit provides the decoded value (v) and the actual length (m) of the input code, where m is less than or equal to n . It also provides the length (a) of a variable length bit field, where (a) is greater than or equal to 0. The variable-length bit field is not Huffman encoded and follows immediately the Huffman code. The n -long bit field

presented to the input of the decoding unit may be longer than or equal to the actual code. The decoding unit determines the actual length of the code (m) and passes it together with the length of the additional bits (a) to a control block. The control block calculates a shift value (a+m) driving the data and marker shifting blocks to shift the input data for the next decoding cycle.

The apparatus of the invention can comprise any combinatorial decoding unit, including ROM, RAM, PLA or anything else based as long as it provides a decoded value, the actual length of the input code, and the length of the following not Huffman encoded bit field within a given time frame.

In the illustrated embodiment, the decoding unit outputs predictively encoded DC difference values and AC run-length values as defined in JPEG standard. The not Huffman encoded bit fields, which are extracted from the input data simultaneously with decoded values, represent additional bits determining the value of the DC and AC coefficients as defined in JPEG standard. Another kind of not Huffman encoded bit fields, which are removed from the data present in the data decode register, are padding bits as defined in JPEG standard that precede byte-aligned markers in the original input data stream. These bits are detected by the control block by checking the contents of a padding zone of the data register. The padding zone comprises up to k most significant bits of the data register and is indicated by the presence of a marker bit within k most significant bits of the marker register, position of said marker bit limiting the length of the padding zone. If all the bits in the padding zone are identical (and equal to 1s in case of JPEG standard), they are considered as padding bits and are removed from the data register accordingly without being decoded. The contents of the data and marker registers are then adjusted for the next decoding cycle.

The exemplary apparatus comprises an output block that handles formatting of the outputted data according to the requirements of the preferred embodiment of the invention. It outputs the decoded values together with the corresponding not variable length encoded bit fields, such as

additional bits in JPEG, and a signal indicating position of any inputted byte aligned and not encoded bit fields, such as markers in JPEG, with respect to the decoded values.

Data being decoded by the JPEG coder 241 (Fig. 2) is JPEG
 5 compatible and comprises variable length Huffman encoded codes interleaved with variable length not encoded bit fields called "additional bits", variable length not encoded bit fields called "padding bits" and fixed length, byte aligned and not encoded bit fields called "markers", "stuff bytes" and "fill bytes". Fig. 82 shows a representative example of input data.

10 The overall structure and the data flow in the Huffman decoder of the JPEG coder 241 is presented in Fig. 83 and Fig. 84, where Fig. 83 illustrates the architecture of the Huffman decoder of the JPEG data in more detail. The stripper 1171 removes marker codes (code $FFXX_{hex}$, XX being non zero), fill bytes (code FF_{hex}) and stuff bytes (code 00_{hex} following code FF_{hex}), that
 15 is all byte aligned components of the input data, which are presented to the stripper as 32 bit words. The most significant bit of the first word to be processed is the head of the input bit stream. In the stripper 1171, the byte aligned bit fields are removed from each input data word before the actual decoding of Huffman codes takes place in the downstream parts of the
 20 decoder.

The input data arrives at the stripper's 1171 input as 32-bit words, one word per clock cycle. Numbering of the input bytes 1211 from 0 to 3 is shown in Fig. 85. If a byte of a number (i) is removed because it is a fill byte, a stuff byte or belongs to a marker, the remaining bytes of numbers (i-1) down
 25 to 0 are shifted to the left on the output of the stripper 1171 and take numbers (i) down to 1. Byte 0 becoming a "don't care" byte. Validity of bytes outputted by the stripper 1171 is also coded by means of separate output tags 1212 as shown in Fig. 85. The bytes which are not removed by the stripper 1171 are left aligned on the stripper's output. Each byte on the
 30 output has a corresponding tag indicating if the corresponding byte is valid (i.e.

passed on by the stripper 1171), or invalid (i.e. removed by the stripper 1171) or valid and following a removed marker. The tags 1212 control loading of the data bytes into the data register 1182 through the data shifter and loading of marker positions into the marker register 1183 through the marker shifter. The same scheme applies if more than one byte is removed from the input word: all the remaining valid bytes are shifted to the left and the corresponding output tags indicate validity of the output bytes. Fig. 85 provides examples 1213 of output bytes and output tags for various example combinations of input bytes.

Returning to Fig. 83, the role of the preshifter and postshifter blocks 1172, 1173, 1180, 1181 is to assure loading of the data into the corresponding data register 1182 and marker register 1183 in a contiguous way whenever there is enough room in the data register and the marker register. The data shifter and the marker shifter blocks, which consist of the respective pre- and postshifters, are identical and identically controlled. The difference is that while the data shifter handles data passed by the stripper 1171, the marker shifter handles the tags only and its role is to pass marker positions to the output of the decoder in a way synchronous with the decoded Huffman values. The outputs of the postshifters 1180, 1181 feed directly to the respective registers 1182, 1183, as shown in Fig. 83.

In the data preshifter 1172, as also shown in Fig. 86, data arriving from the stripper 1171 is firstly extended to 64 bits by appending 32 zeroes to the least significant bit 1251. Then the extended data is shifted in a 64 bit wide barrel shifter 1252 to the right by a number of bits currently present in the data register 1182. This number is provided by the control logic 1185 which keeps track of how many valid bits are there in the data 1182 and marker 1183 registers. The barrel shifter 1252 then presents 64 bits to the multiplexer block 1253, which consists of 64 2x1 elementary multiplexers 1254. Each elementary 2x1 multiplexer 1254 takes as inputs one bit from the barrel shifter 1252 and one bit from the data register 1182. It passes the

data register bit to the output when this bit is still valid in the data register. Otherwise, it passes the barrel shifter's 1252 bit to the output. The control signals to all the elementary multiplexers 1254 are decoded from a control block's shift control 1 signals as shown in Fig. 86, which are also shown in Fig. 87 as preshifter control bits 0...5 of register 1223. The outputs of the elementary multiplexers 1254 drive a barrel shifter 1255. It shifts left by the number of bits provided on a 5 bit control signal shift control 2 as shown in Fig 86. These bits represent the number of bits consumed from the data register 1182 by the decoding of the current data, which can be either the length of the currently decoded Huffman code plus the number of the following additional bits, or the number of padding bits to be removed if padding bits are currently being detected, or zero if the number of valid data bits in the data register 1182 is less then the number of bits to be removed. In this way, the data appearing on the output of barrel shifter 1255 contains new data to be loaded into the data register 1182 after a single decoding cycle. The contents of the data register 1182 changes in such a way that the leading (most significant) bits are shifted out of the register as being decoded, and 0, 8, 16, 24 or 32 bits from the stripper 1171 are added to the contents of the data register 1182. If there are not enough bits in the data register 1182 to decode them, data from the stripper 1171, if available, is still loaded in the current cycle. If there is no data available from the stripper 1171 in the current cycle, the decoded bits from the data register 1182 are still removed if there is a sufficient amount of them, otherwise the content of the data register 1182 does not change.

The marker preshifter 1173, postshifter 1181 and the marker register 1183 are units identical to the data preshifter 1172, data postshifter 1180 and the data register 1182, respectively. The data flow inside units 1173, 1181 and 1183 and among them is also identical as the data flow among units 1172, 1180 and 1182. The same control signals are provided to both sets of units by the control unit 1185. The difference is only in the type of

data on the inputs of the marker preshifter 1173 and data preshifter 1172, as well as in how the contents of the marker register 1183 and the data register 1182 are used. As shown in Fig. 88, tags 1261 from the stripper 1171 come as eight bit words, which provide two bits for each corresponding byte of data going to the data register 1182. According to the coding scheme shown in Fig. 85, an individual two bit tag indicating valid and following a marker byte has 1 on the most significant position. Only this most significant position of each of the four tags delivered by the stripper 1171 simultaneously is driven to the input 1262 of the marker preshifter 1173. In this way, on the input to the marker preshifter there may be bits set to 1 indicating positions of the first encoded data bits following markers. At the same time, they mark the positions of the first encoded data bits in the data register 1182 which follow a marker. This synchronous behavior of the marker position bits in the marker register 1183 and the data bits in the data register 1182 is used in the control block 1185 for detection and removal of padding bits, as well as for passing marker positions to the output of the decoder in a way synchronous with the decoded data. As mentioned, the two preshifters (data 1172 and marker 1173), postshifters (data 1180 and marker 1181) and registers (data 1182 and marker 1183) get the same control signals which facilitates fully parallel and synchronous operation.

The decoding unit 1184, also shown in Fig. 89 gets the sixteen most significant bits of the data register 1182 which are driven to a combinatorial decoding unit 1184 for extraction of a decoded Huffman value, the length of the present input code being decoded and the length of the additional bits following immediately the input code (which is a function of the decoded value). The length of the additional bits is known after the corresponding preceding Huffman symbol is decoded, so is the starting position of the next Huffman symbol. This effectively requires, if speed of one value decoded per clock cycle is to be maintained, that decoding of a Huffman value is done in a combinatorial block. Preferably, the decoding unit comprizes four PLA style

decoding tables hardwired as a combinatorial block taking a 16-bit token on input from the data register 1182 and producing a Huffman value (8 bits), the length of the corresponding Huffman-encoded symbol (4 bits) and the length of the additional bits (4 bits) as illustrated in Fig. 89.

5 Removal of padding bits takes place during the actual decoding when a sequence of padding bits is detected in the data register 1182 by a decoder of padding bits which is part of the control unit 1185. The decoder of padding bits operates as shown in Fig. 90. Eight most significant bits of the marker register 1183, 1242 are monitored for presence of a marker position bit. If a
10 marker position bit is detected, all the bits in the data register 1182, 1241 which correspond to, that is have the same positions as, the bits preceding the marker bit in the marker register 1242 are recognized as belonging to a current padding zone. The content of the current padding zone is checked by the detector of padding bits 1243 for 1's. If all the bits in the current padding
15 zone are 1's, they are recognized as padding bits and are removed from the data register. Removal is done by means of shifting of the contents of the data register 1182, 1241 (and at the same time the marker register 1183, 1242) to the left using the respective shifters 1172, 1173, 1180, 1181 in one clock cycle, as in normal decode mode with the difference that no decoded value is
20 outputted. If not all the bits in the current padding zone are 1's, a normal decode cycle is performed rather than a padding bits removal cycle. Detection of padding bits takes place each cycle as described, in case there are some padding bits in the data register 1182 to be removed.

The control unit 1185 is shown in detail in Fig. 87. The central part
25 of the control unit is the register 1223 holding the current number of valid bits in the data register 1182. The number of valid bits in the marker register 1183 is always equal to the number of valid bits in the data register 1182. The control unit performs three functions. Firstly, it calculates a new number of bits in the data register 1182 to be stored in the register 1223. Secondly,
30 it determines control signals for the shifters 1172, 1173, 1180, 1181, 1186,

1187 decoding unit 1184, and the output formatter 1188. Finally, it detects padding bits in the data register 1182, as described above.

The new number of bits in the data register 1182 (*new_nob*) is calculated as the current number of bits in the data register 1182 (*nob*) plus the number of bits (*nos*) available for loading from the stripper 1171 in the current cycle, less the number of bits (*nor*) removed from the data register 1182 in the current cycle, which is either a decode cycle or a padding bits removal cycle. The new number of bits is calculated as follows:

$$\text{new_nob} = \text{nob} + \text{nos} - \text{nor}$$

The respective arithmetic operations are done in adder 1221 and subtractor 1222. It should be noted that (*nos*) can be 0 if there is no data available from the stripper 1171 in the current cycle. Also, (*nor*) can be 0 if there is no decoding done in the current cycle because of shortage of bits in the data register 1182, which means there are less bits in the data register than the sum of the current code length and the following additional bits length as delivered by the control unit 1185. The value (*new_nob*) may exceed 64 and block 1224 checks for this condition. In such a case, the stripper 1171 is stalled and no new data is loaded. Multiplexer 1233 is used for zeroing the number of bits to be loaded from the stripper 1171. A corresponding signal for stalling the stripper 1171 is not shown. Signal "padding cycle" driven by decoder 1231 controls multiplexer 1234 to select either the number of padding bits or the number of decoded bits (that is the length of code bits plus additional bits) as number of bits to be removed (*nor*). If the number of the decoded bits is greater than the number (*nob*) of the bits in the data register, which is checked in comparator 1228, the effective number of bits to shift as provided for multiplexer 1234 is set to zero by a complex NAND gate 1230. As a result, (*nor*) is set to zero and no bits are removed from the data register. The output of multiplexer 1234 is also used to control postshifters 1182 and 1183. The width of the data register 1182 must be chosen in a way preventing a deadlock situation. This means that at any time either there

needs to be room in the data register to accommodate the maximum number of bits available from the stripper 1171 or sufficient number of valid bits to be removed as a result of a decode or a padding of bits removed cycle.

Calculation of the number of bits to be removed in a decode cycle is performed by adder 1226. Its operands come from the combinatorial decoding unit 1184. As the code length of 16 bits is coded as "0000" by the decoding unit, "or_reduce" logic 1225 provides encoding of "0000" into "10000", yielding a correct unsigned operand. This operand together with the output of subtractor 1227 provide control signals to the output formatting shifters 1186 and 1187.

Block 1229 is used for detection of EOI (End Of Image) marker position. The EOI marker itself is removed by the stripper 1171, but there can be some padding bits which are the very last bits of the data and which used to precede the EOI marker before its removal in the stripper 1171. The comparator 1229 checks if the number of bits in the data register 1182, stored in register 1223 is less than eight. If it is, and there is no more data to come from the stripper 1171 (that is the data register 1182 holds all the remaining bits for of the data unit being decoded), the remaining bits define the size of the padding zone before the removed EOI marker. Further handling of the padding zone and possible removal of padding bits is identical to the procedure applied in case of padding bits before RST markers, which has been described before.

Barrel shifters 1186, 1187 and output formatter 1188 play a support role and depending on the embodiment may have a different implementation or may not be implemented at all. Control signals to them come from the control unit 1185, as described above. The ab_preshifter (additional bits preshifter) 1186 takes 32 bits from the data register as input and shifts them to the left by the length of the Huffman code being presently decoded. In this way, all the additional bits following the code being presently decoded appear left aligned on the output of the barrel shifter 1186 which is also the input to

the barrel shifter 1187. The ab_postshifter (additional bits postshifter) 1187 adjusts the position of the additional bits from left aligned to right aligned in an 11 bit field, as used in the output format of the data and shown in Fig. 91. The additional bits field extends from bit 8 to bit 18 in the output
 5 word format 1196 and some of the most significant bits may be invalid, depending on the actual number of the additional bits. This number is encoded on bits 0 to 3 of 1196, as specified by the JPEG standard. If a different format of the output data is adopted, the barrel shifters 1186 and 1187 and their functionality may change accordingly.

10 The output formatter block 1188 packs the decoded values, which in JPEG standard are DC and AC coefficients, (1196, bits 0 to 7) and a DC coefficient indicator (1196, bit 19) passed by the control unit 1185 together with the additional bits (1196, bits 8 to 18) passed by the ab_postshifter 1187 and the marker position bit (1196, bit 23) from the marker register 1183
 15 into words according to the format presented in Fig. 91. The output formatter 1188 also handles any particular requirements as to the output interface of the decoder. The implementation of the output formatter is normally expected to change if the output interface changes as a result of different requirements.

20 The foregoing described Huffman decoder provides a highly effective form of decoding providing a high speed decoding operation.

3.17.8 Image Transformation Instructions

These instructions implement general affine transformations of source images. The operation to construct a portion of a transformed image
 25 falls generally into two broad areas. These include firstly working out which parts of the source image are relevant to constructing the current output scanline and, if necessary, decompressing them. The second step normally comprizes necessary sub-sampling and/or interpolation to construct the output image on a pixel by pixel basis.

Turning to Fig. 92, there is illustrated a flow chart of the steps required 720 to calculate the value of a destination pixel assuming that the appropriate sections of the source image have been decompressed. Firstly, the relevant sub-sampling, if present, must be taken into account 721. Next, two processes are normally implemented, one involving interpolation 722 and the other being sub-sampling. Normally interpolation and sub-sampling are alternative steps, however in some circumstances interpolation and sub-sampling may be used together. In the interpolation process, the first step is to find the four surrounding pixels 722, then determine if pre-multiplication is required 723, before performing bilinear interpolation 724. The bilinear interpolation step 724 is often computationally intensive and limits the operation of the image transformation process. The final step in calculating a destination pixel value is to add together the possibly bilinear interpolated sub-samples from the source image. The added together pixel values can be accumulated 727 in different possible ways to produce destination image pixels of 728.

The instruction word encoding for image transformation instructions is as illustrated in Fig. 93 with the following interpretation being placed on the minor opcode fields.

Table 19: Instruction Word - Minor Opcode Fields

Field	Description
S	0 = bi-linear interpolation is used on the four surrounding source image pixels to determine the actually sampled value 1 = sampled value is snapped to the closest source image pixel value
off[3:0]	0 = do not apply the offset register (mdp_por) to the corresponding channel 1 = apply the offset register (mdp_por) to the corresponding channel
P	0 = do not pre-multiply source image pixels 1 = pre-multiply source image pixels
C	0 = do not clamp output values 1 = clamp output underflows to 0x00 and overflows to 0xFF
A	0 = do not take absolute value of output values 1 = take absolute value of output values before wrapping or clamping

The instruction operand and result fields are interpreted as follows:

5

Table 20: Instruction Operand and Results Word

Operand	Description	Internal Format	External Format
Operand A	kernel descriptor	-	short or long kernel descriptor table
Operand B	Source Image Pixels	other	image table format
Operand C	unused	-	-
Result	pixels	pixles	packed stream, unpacked bytes

Operand A points to a data structure known as a "kernel descriptor" that describes all the information required to define the actual transformation. This data structure has one of two formats (as defined by the L bit in the A descriptor). Fig. 94 illustrates the long form of kernel

10

descriptor coding and Fig. 95 illustrates the short form of encoding. The kernel descriptor describes:

1. Source image start co-ordinates 730 (unsigned fixed point, 24.24 resolution). Location (0,0) is at the top left of the image.
- 5 2. Horizontal 731 and vertical 732 (sub-sample) deltas (2's complement fixed point, 24.24. resolution)
3. A 3 bit **bp** field 733 defining the location of the binary point within the fixed point matrix co-efficients as described hereinafter.
4. Accumulation matrix co-efficients 735 (if present). These are of
10 "variable" point resolution of 20 binary places (2's complement), with the location of the binary point implicitly specified by the **bp** field.
5. An **rl** field 736 that indicates the remaining number of words in the kernel descriptor. This value is equal to the number of rows times the number of columns minus 1.
- 15 The kernel co-efficients in the descriptor are listed row by row, with elements of alternate rows listed in reverse direction, thereby forming a zig zag pattern.

Turning now to Fig. 96, the operand B consists of a pointer to an index table indexing into scan lines of a source image. The structure of the index
20 table is as illustrated in Fig. 96, with the operand B 740 pointing to an index table 741 which in turn points to scan lines (eg. 742) of the required source image pixels. Typically, the index table and the source image pixels are cacheable and possibly located in the local memory.

The operand C stores the horizontal and vertical sub-sample rate.
25 The horizontal and vertical sub-sample rates are defined by the dimensions of the sub-sample weight matrix which are specified if the C descriptor is present. The dimensions of the matrix *r* and *c* are encoded in the data word of the image transformation instruction as illustrated in Fig. 97.

Channel N of a resultant pixel $P[N]$ is calculated in accordance with
30 the following equation:

$$p[n] = (l.offset[n] \cdot mdp_{por}:0000) + \sum_r \sum_c w_{r,c} \cdot s(x+r\Delta x, y+c\Delta y)[n]$$

Internally, the accumulated value is kept to 36 binary places per channel. The
 5 location of the binary point within this field is specified by the BP field. The BP field indicates
 the number of leading bits in the accumulated result to discard. The 36 bit accumulated value
 is treated as a signed 2's complement number and is clamped or wrapped as specified. In Fig.
 98, there is illustrated an example of the interpretation of the BP field in co-efficient encoding.

3.17.9 Convolution Instructions

10 Convolutions, as applied to rendering images, involves applying a two
 dimensional convolution kernel to a source image to produce a resultant
 image. Convolution is normally used for such matters as edge sharpening or
 indeed any image filter. Convolutions are implemented by the co-processor
 224 in a similar manner to image transformations with the difference being
 15 that, in the case of transformations the kernel is translated by the width of
 the kernel for each output pixel, in the case of convolutions, the kernel is
 moved by one source pixel for each output pixel.

If a source image has values $S(x,y)$ and a $n \times m$ convolution kernel has
 values $C(x,y)$, then the n th channel of the convolution $H[n]$ of S and C is given
 20 by:

$$H(x,y)[n] = (l.offset[n] \cdot mdp_{por}:0000) + \sum_i \sum_j S(x+i, y+j) \cdot C(i,j)[n]$$

where $i \in [0,c]$ and $j \in [0,r]$.

25 The interpretation of the offset value, the resolution of intermediate results and the
 interpretation of the bp field are the same as for Image Transformation instructions.

In Fig. 99, there is illustrated an example of how a convolution kernel 750 is applied
 to a source image 751 to produce a resultant image 752. Source image address generation and
 output pixel calculations are performed in a similar manner to that for image transformation
 30 instructions. The instruction operands take a similar form to image transformations. In Fig.

100, there is illustrated the instruction word encoding for convolution instructions with the following interpretation being applied to the various fields.

Table 21: Instruction Word

Field	Description
d	
S	0 = bi-linear interpolation is used on the four surrounding source image pixels to determine the actually sampled value 1 = sampled value is snapped to the closest source image pixel value
C	0 = do not clamp resultant vector values 1 = clamp result vector values: underflow to 0x00, overflow to 0xFF
P	0 = do not pre-multiply input pixels 1 = pre multiply input pixels
A	0 = do not take absolute value of output values 1 = take absolute value of output values before wrapping or clamping
off[3:0]	0 = do not apply the offset register to this channel 1 = apply the offset register to this channel

5

3.17.10 Matrix Multiplication

Matrix multiplication is utilized for many things including being utilized for color space conversion where an affine relationship exists between two color spaces. Matrix multiplication is defined by the following equation:

10

$$\begin{bmatrix} r_x \\ r_y \\ r_z \\ r_o \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ a_z \\ a_o \\ 1 \end{bmatrix}$$

The matrix multiplication instruction operands and results have the following format:

Table 22: Instruction Operand and Results Word

Operand	Description	Internal Format	External Format
Operand A	source image pixels	pixels	packed stream
Operand B	matrix co-efficients	other	image table format
Operand C	unused	-	-
Result	pixels	pixels	packed stream, unpacked bytes

The instruction word encoding for matrix multiplication instructions as illustrated in Fig. 101 with the following table summarising the minor
 5 opcode fields.

Table 23: Instruction Word

Field	Description
C	0 = do not clamp resultant vector values. 1 = clamp resultant vector values: underflow to 0x00, overflow to 0xFF
P	0 = do not pre-multiply input pixels 1 = pre-multiply input pixels
A	0 = do not take absolute value of output values 1 = take absolute value of output values before wrapping or clamping

3.17.11 Halftoning

10 The co-processor 224 implements a multi-level dither for halftoning. Anything from 2 to 255 is a meaningful number of halftone levels. Data to be halftoned can be either bytes (ie. unmeshed or one channel from meshed data) or pixels (ie. meshed) as long as the screen is correspondingly meshed or unmeshed. Up to four output channels (or four bytes from the same channel)
 15 can be produced per clock, either packed bits (for bi-level halftoning) or codes

(for more than two output levels) which are either packed together in bytes or unpacked in one code per byte.

The output half-toned value is calculated using the following formula:

$$(p \times (l - 1) + d) / 255$$

5 Where p is the pixel value ($0 \leq p \leq 255$), l is the number of levels ($2 \leq l \leq 255$) and d is the dither matrix value ($0 \leq d \leq 254$). The operand encoding is as follows:

Table 24: Instruction Operand and Results Word

Operand	Description	Internal Format	External Format
Operand A	source image pixels	pixels	packed stream
	source image bytes	packed bytes, unpacked bytes	packed stream
Operand B	dither matrix co-efficients	pixels, packed bytes, unpacked bytes	packed stream, unpacked bytes
Operand C	unused	-	-
Result	halftone codes	pixels, packed bytes unpacked bytes	packed stream, unpacked bytes

10 In the instruction word encoding, the minor op code specifies a number of halftone levels. The operand B encoding is for the halftone screen and is encoded in the same way as a compositing tile.

3.17.12 Hierarchial Image Format Decompression

15 Hierarchial image format decompression involves several stages. These stages include horizontal interpolation, vertical interpolation, Huffman decoding and residual merging. Each phase is a separate instruction. In the Huffman decoding step, the residual values to be added to the interpolated values from the interpolation steps are Huffman coded. Hence, the JPEG
20 decoder is utilized for Huffman decoding.

In Fig. 102, there is illustrated the process of horizontal interpolation. The output stream 761 consists of twice as much data as the input stream 762 with the last data value 763 being replicated 764. Fig. 103 illustrates horizontal interpolation by a factor of 4.

5 In the second phase of hierarchial image format decompression, rows of pixels are up sampled by a factor of two or four vertically by linear interpolation. During this phase, one row of pixels is on operand A and the other row is on operand B.

When vertically interpolating, either by a factor of two or four, the
10 output data stream contains the same number of pixels as each input stream. In Fig. 104, there is illustrated an example of vertical interpolation wherein two input data streams 770, 771 are utilized to produce a first output stream 772 having a factor of two interpolation or a second output stream 773 having a factor of 4 interpolation. In the case of pixel interpolation, interpolation
15 occurs separately on each of the four channels of four channel pixels.

The residual merging process involves the bitwise addition of two streams of data. The first stream (operand A) is a stream of base values and the second stream (operand B) is a stream of residual values.

In Fig. 105, there is illustrated two input streams 780, 781 and a
20 corresponding output stream 782 for utilising the process of residual merging.

In Fig. 106 there is illustrated the instruction word encoding for hierarchial image format instructions with the following table providing the relevant details of the minor op code fields.

Table 25: Instruction Word - Minor Opcode Fields

Field	Description
R	0 = interpolation 1 = residual merging
V	0 = horizontal interpolation 1 = vertical interpolation
F	0 = interpolate by a factor of 2 1 = interpolate by a factor of 4
C	0 = do not clamp resultant values 1 = clamp resultant values: underflow to 0x00, overflow to 0xFF

3.17.13 Memory Copy Instructions

These instructions are divided into two specifically disjointed groups.

5 **a. General purpose data movement instructions**

These instructions utilize the normal data flow path through the co-processor 224, comprising the input interface module, input interface switch 252, pixel organizer 246, JPEG coder 241, result organizer 249 and then the output interface module. In this case, the JPEG coder module sends data
10 straight through without applying any operation.

Other instructions include data manipulation operations including:

- packing and unpacking sub-byte values (such as bits, two bit values and four bit values) to a byte
- packing and unpacking bytes within a word
- 15 • aligning
- meshing and unmeshing
- byte lane swapping and duplicating
- memory clearing
- replicating values

20 The data manipulation operation is carried out by a combination of the pixel organizer (on input) and the result organizer (on output). In many cases, these instructions can be combined with other instructions.

b. Local DMA instructions

No data manipulation takes place. As seen in Fig. 2 data transfer occurs (in either direction) between the Local Memory 236 and the Peripheral Interface 237. These instructions are the only ones for which execution can be
 5 overlapped with some other instruction. A maximum of one of these instructions can execute simultaneously with a "non overlapped" instruction.

In memory copy instructions, operand A represents the data to be copied and the result operand represents the target address of the memory copy instructions. For general purpose memory copy instructions, the
 10 particular data manipulation operation is specified by the operand B for input and operand C for output operand words.

3.17.14 Flow Control Instructions

The flow control instructions are a family of instructions that provide control over various aspect of the instruction execution model as described
 15 with reference to Fig. 9. The flow control instructions include both conditional and unconditional jumps enabling the movement from one virtual address to another when executing a stream of instructions. A conditional jump instruction is determined by taking a co-processor or register, masking off any relevant fields and comparing it to given value. This provides for
 20 reasonable generality of instructions. Further, flow control instructions include wait instructions which are typically used to synchronize between overlapped and non-overlapped instructions or as part of micro-programming.

In Fig. 107, there is illustrated instruction when encoding for flow control instructions with the minor opcodes being interpreted as follows:

25

Table 26: Instruction Word - Minor Opcode Fields

Field	Description
type	00 = jump 01 = wait
C	0 = unconditional jump 1 = condition jump
S	0 = use Operand B as Condition Register and Operand C as Condition mask 1 = any interrupt condition set
N	0 = jump if condition is true 1 = dont jump if condition is true
O	0 = wait on non-overlapped instruction to finish 1 = wait on overlapped instruction to finish

In respect of Jump Instructions, the operand A word specified the target address of the jump instruction. If the S bit of the Minor Opcode is set to 0, then operand B specified a co-processor register to use as the source of the condition. The value of the operand B descriptor specifies the address of the register, and the value of the operand B word defines a value to compare the contents of the register against. The operand C word specifies a bitwise mask to apply to the result. That is, the Jump Instruction's condition is true of the bitwise operation:

$$(((\text{register_value xor Operand B}) \text{ and Operand C}) = 0x00000000)$$

Further instructions are also provided for accessing registers for providing full control at the micro programmed level.

3.18 Modules of the Accelerator Card

Turning again to Fig. 2, there will now be provided further separate description of the various modules.

3.18.1 Pixel Organizer

The pixel organizer 246 addresses and buffers data streams from the input interface switch 252. The input data is stored in the pixel organizer's internal memory or buffered to the MUV buffer 250. Any necessary data manipulation is performed upon the input stream before it is delivered to the

main data path 242 or JPEG coder 241 as required. The operating modes of the pixel organizer are configurable by the usual CBus interface. The pixel organizer 246 operates in one of five modes, as specified by a PO_CFG control register. These modes include:

- 5 (a) Idle Mode - where the pixel organizer 246 is not performing any operations.
- (b) Sequential Mode - when input data is stored in an internal FIFO and the pixel organizer 246 sends out requests for data to the input interface switch 252, generating 32 bit addresses for this data.
- 10 (c) Color Space Conversion Mode - when the pixel organizer buffers pixels for color space conversion. In addition, requests are made for interval and fractional values stored in the MUV buffer 250.
- (d) JPEG Compression Mode - when the pixel organizer 246 utilizes the MUV buffer to buffer image data in the form of MCU's.
- 15 (e) Convolution and Image Transformation Mode - when the pixel organizer 246 stores matrix co-efficients in the MUV buffer 250 and passes them, as necessary, to the main data path 242.

The MUV buffer 250 is therefore utilized by the pixel organizer 246 for both main data path 242 and JPEG coder 241 operations. During color
 20 space conversion, the MUV RAM 250 stores the interval and fractional tables and they are accessed as 36 bits of data (four color channels) x (4 bit interval values and 8 bit fractional values). For image transformation and convolution, the MUV RAM 250 stores matrix co-efficients and related configuration data. The co-efficient matrix is limited to 16 rows x 16
 25 columns with each co-efficient being at a maximum 20 bits wide. Only one co-efficient per clock cycle is required from the MUV RAM 250. In addition to co-efficient data, control information such as binary point, source start coordinates and sub-sample deltas must be passed to the main data path 242. This control information is fetched by the pixel organizer 246 before any of the
 30 matrix co-efficients are fetched.

During JPEG compression, the MUV buffer 250 is utilized by the pixel organizer 246 to double buffer MCU's. Preferably, the technique of double buffering is employed to increase the performance of JPEG compression. One half of the MUV RAM 250 is written to using data from the input interface switch 252 while the other half is read by the pixel organizer to obtain data to send to the JPEG coder 241. The pixel organizer 246 is also responsible for performing horizontal sub-sampling of color components where required and to pad MCU's where an input image does not have a size equal to an exact integral number of MCUs.

The pixel organizer 246 is also responsible for formatting input data including byte lane swapping, normalization, byte substitution, byte packing and unpacking and replication operations as hereinbefore discussed with reference to Fig. 32 of the accompanying drawings. The operations are carried out as required by setting the pixel organizers registers.

Turning now to Fig. 108, there is shown the pixel organizer 246 in more detail. The pixel organizer 246 operates under the control of its own set of registers contained within a CBus interface controller 801 which is interconnected to the instruction controller 235 via the global CBus. The pixel organizer 246 includes an operand fetch unit 802 responsible for generating requests from the input interface switch 252 for operand data needed by the pixel organizer 246. The start address for operand data is given by the PO_SAID register which must be set immediately before execution. The PO_SAID register may also hold immediate data, as specified by the L bit in the PO_DMR register. The current address pointer is stored in the PO_CDP register and is incremented by the burst length of any input interface switch request. When data is fetched into the MUV RAM 250, the current offset for data is concatenated with a base address for the MUV RAM 250 as given by the PL_MUV register.

A FIFO 803 is utilized to buffer sequential input data fetched by the operand fetch unit 802. The data manipulation unit 804 is responsible for

implementing for implementing the various manipulations as described with reference to Fig. 32. The output of the data manipulation unit is passed to the MUV address generator 805 which is responsible for passing data to the MUV RAM 250, main data path 242 or JPEG coder 241 in accordance with configuration registers. A pixel organizer control unit 806 is a state machine that generates the required control signals for all the sub-modules in the pixel organizer 246. Included in these signals are those for controlling communication on the various Bus interfaces. The pixel organizer control unit outputs diagnostic information as required to the miscellaneous module 239 according to its status register settings.

Turning now to Fig. 109, there is illustrated the operand fetch unit 802 of Fig. 108 in more detail. The operand fetch unit 802 includes an Instruction Bus address generator (IAG) 810 which contains a state machine for generating requests to fetch operand data. These requests are sent to a request arbiter 811 which arbitrates between requests from the address generator 810 and those from the MUV address generator 805 (Fig. 108) and sends the winning requests to the input (MAG) interface switch 252. The request arbiter 811 contains a state machine to handle requests. It monitors the state of the FIFO via FIFO count unit 814 to decide when it should dispatch the next request. A byte enable generator 812 takes information on the IAG 810 and generates byte enable patterns 816 specifying the valid bytes within each operand data word returned by the input interface switch 252. The byte enabled pattern is stored along with the associated operand data in the FIFO. The request arbiter 811 handles MAG requests before IAG requests when both requests arrive at the same time.

Returning to Fig. 108, the MUV address generator 805 operates in a number of different modes. A first of these modes is the JPEG (compression) mode. In this mode, input data for JPEG compression is supplied by the data manipulation units 804 with the MUV buffer 250 being utilized as a double buffer. The MUV RAM 250 address generator 805 is responsible for

generating the right addresses to the MUV buffer to store incoming data processed by the data manipulation unit 804. The MAG 805 is also responsible for generating read addresses to retrieve color component data from the stored pixels to form 8×8 blocks for JPEG compression. The MAG 805 is also responsible for dealing with the situation when a MCU lies partially on the image. In Fig. 110, there is illustrated an example of a padding operation carried out by the MAG 805.

For normal pixel data, the MAG 805 stores the four color components at the same address within the MUV RAM 250 in four 8 bit rams. To facilitate retrieval of data from the same color channel simultaneously, the MCU data is barrel shifted to the left before it is stored in the MUV RAM 250. The number of bytes the data is shifted to the left is determined by the lowest two bits of the write address. For example, in Fig. 111 there is illustrated the data organization within the MUV RAM 250 for 32 bit pixel data when no sub-sampling is needed. Sub-sampling of input data maybe selected for three or four channel interleaved JPEG mode. In multichannel JPEG compression mode with subsampling operating, the MAG 805 (Fig. 108) performs the sub-sampling before the 32 bit data is stored in the MUV RAM 250 for optimal JPEG coder performance. For the first four incoming pixels, only the first and fourth channels stored in the MUV RAM 250 contains useful data. The data in the second and third channel is sub-sampled and stored in a register inside the pixel organizer 246. For the next four incoming pixels, the second and third channel are filled with sub-sampled data. In Fig. 112, there is illustrated an example of MCU data organization for multi-channel sub-sampling mode. The MAG treats all single channel unpacked data exactly the same as multi-channel pixel data. An example of single channel packed data as read from the MUV RAM is illustrated in Fig. 113.

While the writing process is storing an incoming MCU into the MUV RAM, the reading process is reading 8×8 blocks out of the MUV RAM. In general, the blocks are generated by the MAG 805 by reading the data for each

channel sequentially, four co-efficients at the time. For pixel data and unpacked input data, the stored data is organized as illustrated in Fig. 111. Therefore, to compose one 8×8 block of non-sampled pixel data, the reading process reads data diagonally from the MUV RAM. An example of this process is illustrated in Fig. 114, which shows the reading sequence for four channel data, the form of storage in the MUV RAM 250 assisting to read multiple values for the same channel simultaneously.

When operating in color conversion mode, the MUV RAM 250 is used as a cache to hold the interval and fractional values and the MAG 805 operates as a cache controller. The MUV RAM 250 caches values for three color channels with each color channel containing 256 pairs of four bit interval and fractional values. For each pixel output via the DMU, the MAG 805 is utilized to get the values from the MUV RAM 250. Where the value is not available, the MAG 805 generates a memory read request to fetch the missing interval and fractional values. Instead of fetching one entry in each request, multiple entries are fetched simultaneously for better utilization of bandwidth.

For image transformation and convolution, the MUV RAM 250 stores the matrix co-efficients for the MDP. The MAG cycles through all the matrix co-efficient stored in the MUV RAM 250. At the start of an image transformation and convolution instruction, the MAG 805 generates a request to the operand fetch unit to fetch the kernel description "header" (Fig. 94) and the first matrix co-efficient in a burst request.

Turning now to Fig. 115, there is illustrated the MUV address generator (MAG) 805 of Fig. 108 in more detail. The MAG 805 includes an IBus request module 820 which multiplexers IBus requests generated by an image transformation controller (ITX) 821 and a color space conversion (CSC) controller 822. The requests are sent to the operand fetch unit which services the request. The pixel organizer 246 is only operated either in image transformation or color space conversion mode. Hence, there is no

arbitration required between the two controllers 821, 822. The IBus request module 820 derives the information for generating a request to the operand fetch unit including the burst address and burst length from the relevant pixel organizer registers.

5 A JPEG controller 824 is utilized when operating in JPEG mode and comprises two state machines being a JPEG write controller and a JPEG read controller. The two controllers operate simultaneously and synchronize with each other through the use of internal registers.

10 In a JPEG compression operation, the DMU outputs the MCU data which is stored into the MUV RAM. The JPEG Write Controller is responsible for horizontal padding and control of pixel subsampling, while the JPEG Read Controller is responsible for vertical padding. Horizontal padding is achieved by stalling the DMU output, and vertical padding is achieved by reading the previously read 8x8 block line.

15 The JPEG Write Controller keeps track of the position of the current MCU and DMU output pixel on the source image, and uses this information to decide when the DMU has to be stalled for horizontal padding. When a MCU has been written into the MUV RAM 250, the JPEG Write Controller sets/resets a set of internal registers which indicates the MCU is on the right
20 edge of the image, or is at the bottom edge of the image. The JPEG Read Controller then uses the content of these registers to decide if it is required to perform vertical padding, and if it has read the last MCU on the image.

The JPEG Write Controller keeps track of DMU output data, and stores the DMU output data into the MUV RAM 250.

25 The controller uses a set of registers to record the current position of the input pixel. This information is used to perform horizontally padding by stalling the DMU output.

When a complete MCU has been written into the MUV RAM 250, the controller writes the MCU information into JPEG-RW-IPC registers which is
30 later used by the JPEG Read Controller.

The controller enters the SLEEP state after the last MCU has been written into the MUV RAM 250. The controller stays in this state until the current instruction completes.

The JPEG Read Controller read the 8x8 blocks from the MCUs stored
 5 in the MUV RAM 250. For multi-channel pixels, the controller reads the MCU several times, each time extracting a different byte from each pixel stored in the MUV RAM.

The controller detects if it needs to perform vertical padding using the information provided by the JPEG-RW-IPC. Vertical padding is achieved by
 10 re-reading the last 8-bytes read from the MUV RAM 250.

The Image Transformation Controller 821 is responsible for reading the kernel discriptor from the IBus and passes the kernel header to the MDP 242, and cycles through the matrix co-efficients as many times as specified in the polen register. All data output by the PO 246 in an image
 15 transformation and Convolution instruction are fetched directly from the IBus and not passed through the DMU.

The top eight bits of the first matrix co-efficient fetched immediately after the kernel header contains the number of remaining matrix co-efficients to be fetched.
 The kernel header is passed to the
 20 MDP directly without modifications, whilst the matrix co-efficients are sign extended before they are passed to the MDP.

The pixel sub-sampler 825 comprizes two identical channel sub-samplers, each operating on a byte from the input word. When the relevant configuration register is not asserted, the pixel sub-sampler copies its input to
 25 its output. When the configuration register is asserted, the sub-sampler sub-samples the input data either by taking the average or by decimation.

An MUV multiplexer module 826 selects the MUV read and write signals from the currently active controller. Internal multiplexers are used to select the read addresses output via the various controllers that utilize the
 30 MUV RAM 250. An MUV RAM write address is held in an 8 bit register in

an MUV multiplexer module. The controllers utilising the MUV RAM 250, load the write address register in addition to providing control for determining a next MUV RAM address.

A MUV valid access module 827 is utilized by the color space
 5 conversion controller to determine if the interval and fractional values for a current pixel output by the data manipulation unit is available in the MUV RAM 250. When one or more color channels are missing, the MUV valid access module 827 passes the relevant address to the IBus request module 820 for loading in burst mode, interval and fractional values. Upon servicing
 10 a cache miss, the MUV valid access module 827 sets internal validity bits which map the set of interval and fractional values fetched so far.

A replicate module 829 replicates the incoming data, the number of times as specified by an internal pixel register. The input stream is stalled while the replication module is replicating the current input word. A PBus
 15 interface module 630 is utilized to re-time the output signals of the pixel organizer 246 to the main data path 242 and JPEG coder 241 and vice versa. Finally, a MAG controller 831 generates signals for initiating and shutting down the various sub-modules. It also performs multiplexing of incoming PBus signals from the main data path 242 and JPEG coder 241.

20 3.18.2 MUV Buffer

Returning to Fig. 2, it will be evident from the foregoing discussion that the pixel organizer 246 interacts with the MUV buffer 250.

The reconfigurable MUV buffer 250 is able to support a number of operating modes including the single lookup table mode (mode0), multiple
 25 lookup table mode (mode1), and JPEG mode (mode2). A different type of data object is stored in the buffer in each mode. For instance, the data objects that are stored in the buffer can be data words, values of a multiplicity of lookup tables, single channel data and multiple channel pixel data. In general, the data objects can have different sizes. Furthermore, the data
 30 objects stored in the reconfigurable MUV buffer 250 can be accessed in

substantially different ways which is dependent on the operating mode of the buffer.

To facilitate the different methods needed to store and retrieve different types of data objects, the data objects are often encoded before they are stored. The coding scheme applied to a data object is determined by the size of the data object, the format that the data objects are to be presented, how the data objects are retrieved from the buffer, and also the organization of the memory modules that comprize the buffer.

Fig 116 is a block diagram of the components used to implement the reconfigurable MUV buffer 250. The reconfigurable MUV buffer 250 comprizes an encoder 1290, a storage device 1293, a decoder 1291, and a read address and rotate signal generator 1292. When a data object arrives from an input data stream 1295, the data object may be encoded into an internal data format and placed on the encoded input data stream 1296 by the encoder 1290. The encoded data object is stored in the storage device 1293.

When decoding previously stored data objects, an encoded data object is read out of the storage device via encoded output data stream 1297. The encoded data object in the encoded output data stream 1297 is decoded by a decoder 1291. The decoded data object is then presented at the output data stream 1298.

The write addresses 1305 to the storage device 1293 are provided by the MAG 805 (Fig 108). The read addressses 1299, 1300 and 1301 are also provided by the MAG 805 (Fig 108), and translated and multiplexed to the storage device 1293 by the Read Address and Rotate Signal Generator 1292, which also generates input and output rotate control signals 1303 and 1304 to the encoder and decoder respectively. The write enable signals 1306 and 1307 are provided by an external source. An operating mode signal 1302, which is provided by means of the controller 801(Fig 108), is connected to the encoder 1290, the decoder 1291, the Read Address and Rotate Signal Generator 1292, and the storage device 1293. An increment signal 1308

increments internal counter(s) in the read address and rotate signal generator and may be utilized in JPEG mode (mode2).

Preferably, when the reconfigurable MUV buffer 250 is operating in the single lookup table mode (mode0), the buffer behaves substantially like a single memory module. Data objects may be stored into and retrieved from the buffer in substantially the same way used to access memory modules.

When the reconfigurable MUV buffer 250 is operating in the multiple lookup table mode (mode 1), the buffer 250 is divided into a plurality of tables with up to three lookup tables may be stored in the storage device 1293. The lookup tables may be accessed separately and simultaneously. For instance, in one example, interval and fraction values are stored in the storage device 1293 in the multiple lookup table mode, and the tables are indexed utilizing the lower three bytes of the input data stream 1295. Each of the three bytes are issued to access a separate lookup table stored in the storage device 1293.

When an image undergoes JPEG compression, the image is converted into an encoded data stream. The pixels are retrieved in the form of MCUs from the original image. The MCUs are read from left to right, and top to bottom from the image. Each MCU is decomposed into a number of single component 8x8 blocks. The number of 8x8 blocks that can be extracted from a MCU depends on several factors including: the number of color components in the source pixels, and for a multiple channel JPEG mode, whether subsampling is needed. The 8x8 blocks are then subjected to forward DCT (FDCT), quantization, and entropy encoding. In the case of JPEG decompression, the encoded data are read sequentially from a data stream. The data stream undergoes entropy decoding, dequantization and inverse DCT (IDCT). The output of the IDCT operation are 8x8 blocks. A number of single component 8x8 blocks are combined to reconstruct a MCU. As with JPEG compression, the number of single component 8x8 blocks are dependent on the same factors mentioned above. The reconfigurable MUV buffer 250 may be used in the process to decompose MCUs into a multiplicity of single

component 8x8 blocks, to reconstruct MCUs from a multiplicity of single component 8x8 blocks.

When the reconfigurable MUV buffer 250 is operating in JPEG mode (mode2), the input data stream 1295 to the buffer 250 comprizes pixels for a
 5 JPEG compression operation, or single component data in a JPEG decompression operation. The output data stream 1298 of the buffer 250 comprizes single channel data blocks for a JPEG compression operation, or pixel data in a JPEG decompression operation. In this example, for a JPEG compression operation, an input pixel may comprize up to four channels
 10 denoted Y, U, V and O. When the required number of pixels have been accumulated in the buffer to form a complete pixel block, the extraction of single component data blocks can commence. Each single component data block comprizes data from the like channel of each pixel stored in the buffer. Thus in this example, up to four single component data blocks may be
 15 extracted from one pixel data block. In this embodiment, when the reconfigurable MUV buffer 250 is operating in the JPEG mode (mode2) for JPEG compression, a multiplicity of Minimum Coded Units (MCUs) each containing 64 single or 64 multiple channel pixels may be stored in the buffer, and a multiplicity of 64-byte long single channel component data blocks are
 20 extracted from each MCU stored in the buffer. In this embodiment, for the buffer 1289 operating in the JPEG mode (mode2) for a JPEG decompression operations, the output data stream contains output pixels that have up to four components Y, U, V and O. When the required number of complete single component data blocks have been written into the buffer, the extraction of
 25 pixel data may commence. A byte from up to four single component block corresponding to different color components are retrieved to form an output pixel.

Fig. 117 illustrates the encoder 1290 of Fig. 116 in more detail. For the pixel block decomposition mode only, each input data object is encoded
 30 using a byte-wise rotation before it is stored into the storage device 1293 (Fig.

129). The amount of rotation is specified by the input rotate control signal 1303. As the pixel data has a maximum of four bytes in this example, a 32-bit 4-to-1 multiplexer 1320 and output 1325 is used to select one of the four possible rotated versions of the input pixel. For example, if the four bytes in
 5 a pixel are labelled (3,2,1,0), the four possible rotated versions of this pixel are (3,2,1,0), (0,3,2,1), (1,0,3,2) and (2,1,0,3). The four encoded bytes are output 1296 for storage in the storage device.

When the buffer is placed in an operating mode other than the JPEG mode (mode2), for example, single lookup table mode (mode0) and multiple
 10 lookup table mode (mode1), byte-wise rotation may not be necessary and may not be performed on the input data objects. The input data object is prevented from being rotated in the latter cases by overriding the input rotate control signal with a no-operation value. This value 1323 can be zero. A 2-to-1 multiplexer 1321 produces control signals 1326 by selecting between the
 15 input rotate control signal 1303 and the no-operation value 1323. The current operating mode 1302 is compared with the value assigned to the pixel block decomposition mode to produce the multiplexer select signal 1322. The 4-to-1 multiplexer 1320, which is controlled by signal 1326 selects one of the four rotated version of the input data object on the input data stream 1325,
 20 and produces an encoded input data object on the encoded input data stream 1326.

Fig. 118 illustrates a schematic of a combinatorial circuit which implements the decoder 1291 for the decoding of the encoded output data stream 1297. The decoder 1321 operates in a substantially similar manner
 25 to the encoder. The decoder only operates on the data when the data buffer is in the JPEG mode (mode2). The lower 32-bit of an encoded output data object in the encoded output data stream 1297 is passed to the decoder. The data is decoded using a byte-wise rotation with an opposite sense of rotation to the rotation performed by the encoder 1290. A 32-bit 4-to-1 multiplexer
 30 1330 is used to select one of the four possible rotated version of the encoded

data. For example, if the four bytes in an input pixel are labelled (3,2,1,0), the four possible rotated version of this pixel are (3,2,1,0), (2,1,0,3), (1,0,3,2) and (0,3,2,1). The output rotate control signal 1304 is utilized only when the buffer is in a pixel block decomposition mode, and when overridden by a no-
 5 operation value in other operating modes. The no-operation value utilized 1333 is zero. A 2-to-1 multiplexer 1331 produces signal 1334 by selecting selects between the output rotate control signal 1304 and the no-operation value 1333. The current operating mode 1302 is compared with the value assigned to the pixel block decomposition mode to produce the multiplexer
 10 select signal 1332. The 4-to-1 multiplexer 1330, which is controlled by signal 1334, selects one of the four rotated version of the encoded output data object on the encoded output data stream 1297, and produces an output data object on the output data stream 1298.

Returning to Fig. 116, the method of internal read address generation
 15 used by the circuit is selected by the operating mode 1302 of the reconfigurable MUV buffer 250. For the single lookup table mode (mode0) and multiple lookup table mode (mode1), the read addresses are provided by the MAG 805 (Fig 108) in the form of external read addresses 1299, 1300, and 1301. For the single lookup table mode (mode0), the memory modules 1380,
 20 1381, 1382, 1383, 1384 and 1385 (Fig. 121) of the storage device 1293 operate together. The read address and the write address supplied to the memory modules 1380 to 1385 (Fig. 121) are substantially the same. Hence the storage device 1293 only needs the external circuits to supply one read address and one write address, and uses internal logic to multiplex these
 25 addresses to the memory modules 1380 to 1385 (Fig. 121). For mode0, the read address is supplied by the external read address 1299 (Fig. 116) and is multiplexed to the internal read address 1348 (Fig. 121) without substantial changes. The external read addresses 1300 and 1301 (Fig. 116), and the internal read addresses 1349, 1350 and 1351 (Fig. 121), are not used in
 30 mode0. The write address is supplied by the external write address 1305

(Fig. 116), and is connected to the write address of each memory module 1380 to 1385 (Fig. 121) without substantial modification.

In this example, a design that provides three lookup tables in the multiple lookup table mode (mode 1) is presented. The encoded input data is written simultaneously into all memory modules 1380 to 1385 (Fig. 121), while the three tables are accessed independently, and thus require one index to each of the three tables. Three indices, that is, read addresses to the memory modules 1380 to 1385 (Fig. 121), are supplied to the storage device 1293. These read addresses are multiplexed to the appropriate memory modules 1380 to 1385 using internal logic. In substantially the same manner as in the single lookup table mode, the write address supplied externally is connected to the write address of each of the memory modules 1380 to 1385 without substantial modifications. Hence, for the multiple lookup table mode (mode 1), the external read addresses 1299, 1300 and 1311 are multiplexed to internal read addresses 1348, 1349 and 1350 respectively. The internal read address 1351 is not used in mode 1. The method of generating the internal read addresses need in the JPEG mode (mode 2) is different to the method described above.

Fig. 119 illustrates a schematic of a combinatorial circuit which implements the read address and rotate control signals generation circuit 1292 (Fig. 116), for the reconfigurable data buffer operating in the JPEG mode (mode 2) for JPEG compression. In the JPEG mode (mode 2), the generator 1292 uses the output of a component block counter 1340 and the output of a data byte counter 1341 to compute the internal read addresses to the memory modules comprising the storage device 1293. The component block counter 1340 gives the number of component blocks extracted from a pixel data block, which is stored in the storage device. The number of like components extracted from the pixel data block is given by multiplying the output of the data byte counter 1341 by four. In this embodiment, an internal read address 1348, 1349, 1350 or 1351 for the pixel data block decomposition mode is computed

as follows. The output of the component block counter is used to generate an offset value 1343, 1344, 1345, 1346 or 1347, and the output of the data byte counter 1341 is used to generate a base read address 1354. The offset value 1343 is added 1358 to the base read address 1354 and the sum is an internal
 5 read address 1348 (or 1349, 1350 or 1351). The offset values for the memory modules are in general different for simultaneous read operations performed on multiple memory modules, but the offset value to each memory module is in general substantially the same during the extraction of one component data block. The base addresses 1354 used to compute the four internal read
 10 addresses in the pixel data block decomposition mode are substantially the same. The increment signal 1308 is used as the component byte counter increment signal. The counter is incremented after every successful read operation has been performed. A component block counter increment signal 1356 is used to increment the component block counter 1340, after a complete
 15 single component data block has been retrieved from the buffer.

The output rotate control signal 1304 (Fig. 116) is derived from the output of the component block counter, and the output of the data byte counter, in substantially similar manner to the generation of an internal read address. The output of the component block counter is used to compute a rotation offset
 20 1347. The output rotate control signal 1304 is given by the lowest two bits of the sum of the base read address 1354 and the rotation offset 1355. The input rotate control signal 1303 is simply given by the lowest two bytes of the external write addresses 1305 in this example of the address and rotate control signals generator.

25 Fig. 120 shows another example of the address generator 1292 for reassembling multiple channel pixel data from single component data stored in the reconfigurable MUV buffer 250. In this case, the buffer is operating in the JPEG (mode2) for JPEG decompression operation. In this case, single component data blocks are stored in the buffer, and pixel data blocks are
 30 retrieved from the buffer. In this example, the write address to the memory

modules are provided by the external write address 1305 without substantial changes. The single component blocks are stored in contiguous memory locations. The input rotate control signal 1303 in this example is simply set to the lowest two bits of the write address. A pixel counter 1360 is used to
 5 keep track of the number of pixels extracted from the single component blocks stored in the buffer. The output of the pixel counter is used to generate the read addresses 1348, 1349, 1350 and 1351, and the output rotate control signal 1304. The read addresses are in general different for each memory module that comprize the storage device 1293. In this example, a read
 10 address comprizes two parts, a single component block index 1362, 1363, 1364 or 1365, and a byte index 1361. An offset is added to bit 3 and 4 of the output of the pixel counter to calculate the single component block index for a particular block. The offsets 1366, 1367, 1368 and 1369 are in general different for each read address. Bit 2 to bit 0 of the output of the pixel
 15 counter are used as the byte index 1361 of a read address. A read address is the result of the concatenation of a single component block index 1362, 1363, 1364 or 1365 and a byte index 1361, as illustrated in Fig. 120. In this example, the output rotate control signal 1304 is generated using bit 4 and bit 3 of the output of the pixel counter without substantial change. The
 20 increment signal 1308 is used as the pixel counter increment signal to increment the pixel counter 1360. The pixel counter 1360 is incremented after a pixel has been successfully retrieved from the buffer.

Fig. 121 illustrates an example of a structure of the storage device 1293. The storage device 1293 can comprize three 4-bit wide memory
 25 modules 1383, 1384 and 1385, and three 8-bit wide memory modules 1380, 1381 and 1382. The memory modules can be combined together to store 36-bit words in the single lookup table mode (mode0), 3x12-bit words in the multiple lookup table mode (mode1), and 32-bit pixels or 4x8-bit single component data in JPEG mode (mode2). Typically each memory module is
 30 associated with a different part of the encoded input and output data streams

(1296 and 1297). For example, memory module 1380 has its data input port connected to bit 0 to bit 7 of the encoded input data stream 1296, and its data output port connected to bit 0 to bit 7 of the encoded output data stream 1297. In this example, the write addresses to all the memory modules are connected together, and share substantially the same value. In contrast, the read addresses 1386, 1387, 1388, 1389, 1390 and 1391 to the memory modules of the example illustrated in Fig. 121 are supplied by the read address generator 1292, and are in general different. In the example, a common write enable signal is used to provide the write enable signals to all three 8-bit memory modules, and a second common write enable signal is used to provide the write enable signals to all three 4-bit memory modules.

Fig. 122 illustrates a schematic of a combinatorial circuit used for generating read addresses 1386, 1387, 1388, 1389, 1390 and 1391 for accessing to the memory modules contained in a storage device 1293. Each encoded input data object is broken up into parts, and each part is stored into a separate memory module in the storage device. Hence, typically the write addresses to all memory modules for all operating modes are substantially the same and thus substantially no logic is required to compute the write address to the memory modules. The read addresses in this example, on the other hand, are typically different for different operations, and are also different to each memory module within each operating mode. All bytes in the output data stream 1298 of the reconfigurable MUV buffer 250 must contain single component data extracted from the pixel data stored in the buffer in the JPEG mode (mode2) for JPEG compression, or pixel data extracted from the single component data blocks stored in the buffer in the JPEG mode for JPEG decomposition. The requirements on the output data stream are achieved by providing four read addresses 1348, 1349, 1350 and 1351 to the buffer. In the multiple lookup table mode (mode1), up to three lookup tables are stored in the buffer, and thus only up to three read addresses 1348, 1349 and 1350 are needed to index the three lookup tables.

The read addresses to all memory modules are substantially the same in the single lookup table mode (mode0), and only read address 248 is used in this mode. The example controller circuit shown in Fig. 122 uses the operating mode signals to the buffer, and up to four read addresses, to compute the read address 1386 - 1391 to each of the six memory modules comprising the storage device 1293. The read address generator 1292 takes, as its inputs, the external read addresses 1299, which comprizes external address buses 1348, 1349, 1350 and 1351, and generates the internal read addresses 1386, 1387, 1388, 1389, 1390 and 1391 to the memory modules that comprize the storage device 1293. No manipulation on the external write addresses 1305 is required in the operation of this example.

Fig. 123 illustrates a representation of an example of how 20-bit matrix coefficients may be stored in the buffer 250 when the buffer 250 is operating in single lookup table mode (mode0). In this example, typically no encoding is applied on the data objects stored in the cache when the data objects are written into the reconfigurable MUV buffer. The matrix coefficients are stored in the 8-bit memory modules 1380, 1381 and 1382. Bit 7 to bit 0 of the matrix coefficient are stored in memory module 1380, bit 15 to bit 8 of the matrix coefficient are stored in memory module 1381, and bit 19 to bit 16 of the matrix coefficient are stored in the lower 4 bits of memory module 1382. The data objects stored in the buffer may be retrrieved as many times as required for the rest of the instruction. The write and read addresses to all memory modules involved in the single lookup table mode are substantially the same.

Fig. 124 illustrates a representation of how the table entries are stored in the buffer in the multiple lookup table mode (mode1). In this example, up to three lookup tables may be stored in the buffer, and each lookup table entry comprizes a 4-bit interval value and an 8-bit fraction value. Typically the interval values are stored in the 4-bit memory modules, and the fraction values are stored in the 8-bit memory modules. The three lookup

tables 1410, 1411 and 1412 are stored in the memory banks 1380 and 1383, 1381 and 1384, 1382 and 1385 in the example. The separate write enable control signals 1306 and 1307 (Fig. 121) allow the interval values to be written into the storage device 1293 without affecting the fraction values
 5 already stored in the storage device. In substantially the same manner, the fraction values may be written into storage device without affecting the interval values already stored in the storage device.

Fig. 125 illustrates a representation of how pixel data is stored in the reconfigurable MUV buffer 250 when the JPEG mode (mode2) for
 10 decomposing pixel data blocks into single component data blocks. The storage device 1293 is organized as four 8-bit memory banks, which comprizes the memory modules 1380, 1381, 1382, 1383 and 1384, with 1383 and 1384 used together to operate substantially in the same manner as an 8-bit memory module. Memory module 1385 is not used in the JPEG mode
 15 (mode2). A 32-bit encoded pixel is broken up into four bytes, and each is stored into a different 8-bit memory module.

Fig. 126 illustrates a representation of how the single component data blocks are stored in the storage device 1293 in single component mode. The storage device 1293 is organized as four 8-bit memory banks, which comprizes
 20 the memory modules 1380, 1381, 1382, 1383 and 1384, with 1383 and 1384 used together to operate substantially in the same manner as an 8-bit memory module. A single component block in this example comprizes 64 bytes. A different amount of byte rotation can be applied to each single component block when it is written into the buffer. A 32-bit encoded pixel
 25 data is retrieved by reading from the different single component data block stored in the buffer.

For further details on the organization of the data within the MUV buffer 250 reference is made herein to the section entitled *Pixel Organizer*.

This preferred embodiment has shown that a reconfigurable data
 30 buffer may be used to handle data involved in different instructions. A

reconfigurable data buffer that provides three operating modes has been disclosed. Different address generation techniques may be needed in each operating mode of the buffer. The single look-up table mode (mode0) may be used to store matrix coefficients in the buffer for an image transformation operation. The multiple look-up table mode (mode1) may be used to store a multiplicity of interval and fraction lookup tables in the buffer in a multiple channel color space conversion (CSC) operation. The JPEG mode (mode2) may be used either to decompose MCU data into single component 8x8 blocks, or to reconstruct MCU data from single-component 8x8 blocks, in JPEG compression and decompression operation respectively.

3.18.3 Result Organizer

The MUV buffer 250 is also utilized by the result organizer 249. The result organizer 249 buffers and formats the data stream from either the main data path 242 or the JPEG coder 241. The result organizer 249 also is responsible for data packing and unpacking, denormalization, byte lane swapping and realignment of result data as previously discussed with reference to Fig. 42. Additionally the result organizer 249 transmits its results to the external interface controller 238, the local memory controller 236, and the peripheral interface controller 237 as required.

When operating in JPEG decompression mode, the results organizer 249 utilizes the MUV RAM 250 to double buffer image data produced by the JPEG coder 241. Double buffering increases the performance of the JPEG decompression by allowing data from the JPEG coder 241 to be written to one half of the MUV RAM 250 while at the same time image data presently in the other half of the MUV RAM 250 is output to a desired destination.

The 1, 3 and 4 channel image data is passed to the result organizer 249 during JPEG decompression in a form of 8 x 8 blocks with each block consisting of 8 bit components from the same channel. The result organizer stores these blocks in the MUV RAM 250 in the order provided and then, for multi-channel interleaved images, meshing of the channels is performed when

reading data from the MUV RAM 250. For example, in a three channel JPEG compression based on Y, U, V color space, the JPEG coder 241 outputs three 8 x 8 blocks, the first consisting of Y components, the second made of the U components and the third made up of the V components. Meshing is accomplished by taking one component from each block and constructing the pixel in the form of (YUVX) where X represents an unused channel. Byte swapping may be applied to each output to swap the channels as desired. The result organizer 249 must also do any required sub-sampling to reconstruct chroma-data from decompressed output. This can involve replicating each program channel to produce and an one.

Turning to Fig. 127, there is illustrated the result organizer 249 of Fig. 2 in more detail. The result organizer 249 is based around the usual standard CBus interface 840 which includes a register file of registers to be set for operation of the result organizer 249. The operation of the result organizer 249 is similar to that of the pixel organizer 246, however the reverse data manipulation operations take place. A data manipulation unit 842 performs byte lane swapping, component substitution, component deselection and denormalization operations on data provided by the MUV address generator (MAG) 805. The operations carried out are those previously described with reference to Fig. 42 and operate in accordance with various fields set in internal registers. The FIFO queue 843 provides buffering of output data before it is output via RBus control unit 844.

The RBus control unit 844 is composed of an address decoder and state machines for address generation. The address for the destination module is stored in an internal register in addition to data on the number of output bytes required. Further, an internal RO_CUT register specifies how many output bytes to discard before sending a byte stream on the output bus. Additionally, a RO_LMT register specifies the maximum number of data items to be output with subsequent data bytes after the output limit being ignored. The MAG 805 generates addresses for the MUV RAM 250 during

JPEG decompression. The MUV RAM 250 is utilized to double buffer output from the JPEG decoder. The MAG 805 performs any appropriate meshing of components in the MUV RAM 250 in accordance with an internal configuration register and outputs single channel, three channel or four channel interleaved pixels. The data obtained from the MUV RAM 250 is then passed through the data manipulation unit 842, since byte lane swapping may need to be applied before pixel data is sent to the appropriate destination. When the results organizer 249 is not configured for JPEG mode, the MAG 805 simply forwards data from the PBus receiver 845 straight through to the data manipulation unit 842.

3.18.4 Operand Organizers B and C

Returning again to Fig. 2, the two identical operand organizers 247, 248 perform the function of buffering data from the data cache control 240 and forwarding the data to the JPEG coder 241 or the main data path 242. The operand organizers 247, 248 are operated in a number of modes:

- (a) Idle mode wherein the operand organizer only responds to CBus requests.
- (b) Immediate mode when the data of the current instruction is stored in an internal register of the operand organizer.
- (c) Sequential mode wherein the operand organizer generates sequential addresses and requests data from the data cache controller 240 whenever its input buffer requires filling.

A number of modes of operation of the main data path 242 require at least one of the operand organizers 247, 248 to operate in sequential mode. These modes include compositing wherein operand organizer B 247 is required to buffer pixels which are to be composited with another image. Operand organizer C 248 is used for compositing operations for attenuation of values for each data channel. In halftoning mode, operand organizer B 247 buffers 8 bit matrix co-efficients and in hierarchial image format

decompression mode the operand organizer B 247 buffers data for both vertical interpolation and residual merging instructions.

(d) In constant mode, an operand organizer B constructs a single internal data word and replicates this word a number of times as given by an
5 internal register.

(e) In tiling mode an operand organizer B buffers data that comprises a pixel tile.

(f) In random mode the operand organizer forwards addresses from the MDP 242 or JPEG coder 241 directly to the data cache controller. These
10 addresses are utilized to index the data cache 230.

An internal length register specifies the number of items to be generated by individual operand organizers 247, 248 when operated in sequential/tiling/constant mode. Each operand organizer 247, 248 keeps account of the number of data items processed so far and stops when the count
15 reaches the value specified in its internal register. Each operand organizer is further responsible for formatting input data via byte lane swapping, component substitution, packed/unpacked and normalization functions. The desired operations are configured utilising internal registers. Further, each operand organizer 247, 248 may also be configured to constrict data items.

20 Turning now to Fig. 128, there is illustrated the structure of operand organizers (247, 248) in more detail. The operand organizer 247, 248 includes the usual standard CBus interface and registers 850 responsible for the overall control of the operand organizer. Further, an OBus control unit 851 is provided for connection to the data cache controller 240 and is
25 responsible for performing address generation for sequential/tile/ constant modes, generating control signals to enable communications on the OBus interface to each operand organizer 247, 248 and controlling data manipulation unit operations such as normalization and replication, that require the state to be saved from previous clock cycles of the input stream.
30 When an operand organizer 247, 248 is operating in sequential or tiling mode,

the OBus control unit 851 sends requests for data to the data cache controller 240, the addresses being determined by internal registers.

Each operand organizer further contains a 36 bit wide FIFO buffer 852 used to buffer data from the data cache controller 240 in various modes of operation.

A data manipulation unit 853 performs the same functions as the corresponding data manipulation unit 804 of the pixel organizer 246.

A main data path/JPEG coder interface 854 multiplexes address and data to and from the main data path and JPEG coder modules 242, 241 in normal operating mode. The MDP/JC interface 854 passes input data from the data manipulation units 853 to the main data path and in the process may be configured to replicate this data. When operating in color conversion mode, the units 851, 854 are bypassed in order to ensure high speed access to the data cache controller 240 and the color conversion tables.

3.18.5 Main Data Path Unit

The aspects of the following embodiment relate to an image processor providing a low cost computer architecture capable of performing a number of image processing operations at high speed. Still further, the image processor seeks to provide a flexible computer architecture capable of being configured to perform image processing operations that are not originally specified. The image processor also seeks to provide a computer architecture having a large amount of identical logic, which simplifies the design process and lowers the cost of designing such an architecture.

The computer architecture comprises a control register block, a decoding block, a data object processor, and flow control logic. The control register block stores all the relevant information about the image processing operation. The decoding block decodes the information into configuration signals, which configure an input data object interface. The input data object interface accepts and stores data objects from outside, and distributes these data objects to the data object processor. For some image processing

operations, the input data object interface may also generate addresses for data objects, so that the source of these data objects can provide the correct data objects. The data object processor performs arithmetic operations on the data objects received. The flow control logic controls the flow of data
5 objects within the data object processing logic.

More particularly, the data object processor can comprise a number of identical data object sub-processors, each of which processes part of an incoming data object. The data object sub-processor includes a number of identical multifunctional arithmetic units that perform arithmetic operations
10 on these parts of data objects, post processing logic that processes the outgoing data objects, and multiplexer logic that connects the multifunctional arithmetic units and the post-processing unit together. The multifunctional arithmetic units contain storage for parts of the calculated data objects. The storage is enabled or disabled by the flow control logic. The multifunctional
15 arithmetic units and multiplexer logic are configured by the configuration signals generated by the decoding logic.

Furthermore, the configuration signals from the decoding logic can be overridden by an external programming agent. Through this mechanism any multifunctional blocks and multiplexer logic can be individually configured by
20 an external programming agent, allowing it to configure the image processor to perform image processing operations that are not specified beforehand. These and other aspects of the embodiments of the invention are described in greater detail hereinafter.

Returning to Fig. 2, as noted previously the main data path unit 242 performs all data manipulation operations and instructions other than JPEG data coding. These instructions include compositing, color space conversion, image transformations, convolution, matrix multiplication, halftoning, memory copying and hierarchial image format decompression. The main data path 242 receives pixel and operand data from the pixel organizer 246, and operand organizers 247, 248 and feeds the resultant output to the result organizer 249.

Fig.129 illustrates a block diagram of the main data path unit 242. The main data path unit 242 is a general image processor and includes input interface 1460, image data processor 1462, instruction word register 1464, instruction word decoder 1468, control signal register 1470, register file 1472, and a ROM 1475.

The instruction controller 235 transfers instruction words to the instruction word register 1464 via bus 1454. Each instruction word contains information such as the kind of image processing operation to be executed, and flags to enable or disable various options in that image processing operation. The instruction word is then transferred to the instruction word decoder 1468 via bus 1465. Instruction controller 235 can then indicate to the instruction word decoder 1468 to decode the instruction word. Upon receiving that indication, the instruction decoder 1468 decodes the instruction word into control signals. These control signals are then transferred via bus 1469 to the control signal register 1470. The output of the control signal register is then connected to the input interface 1460 and image data processor 1462 via bus 1471.

To add further flexibility to the main data path unit 242, the instruction controller 235 can also write into the control signal register 1470. This allows anyone who is familiar with the structure of the main data path unit 242 to micro-configure the main data path unit 242 so that the main

data path unit 242 will execute image processing operations that are not be described by any instruction word.

In cases when all the necessary information to perform the desired image processing operation does not fit into the instruction word, the instruction controller 235 can write all the other information necessary to perform the desired image processing operation into some of the selected registers in register file 1472. The information is then transferred to the input interface 1460 and the image data processor 1462 via bus 1473. For some image processing operations, the input interface 1460 may update the contents of selected registers in the register file 1472 to reflect the current status of the main data path unit 242. This feature helps the instruction controller 235 to find out what the problem is when there is a problem in executing an image processing operation.

Once the decoding of the instruction word is finished, and/or the control signal register is loaded with the desired control signals, the instruction controller 235 can indicate to the main data path unit 242 to start performing the desired image processing operation. Once that indication is received, the input interface 1460 begins to accept data objects coming from bus 1451. Depending on the kind of image processing operation performed, the input interface 1460 may also begins to accept operand data coming from operand bus 1452 and/or operand bus 1453, or generates addresses for operand data and receive operand data from operand bus 1452 and/or operand bus 1453. The input interface 1460 then stores and rearranges the incoming data in accordance with the output of the control signal register 1470. The input interface 1460 also generates coordinates to be fetched via buses 1452 and 1453 when calculating such functions as affine image transformation operations and convolution.

The image data processor 1462 performs the major arithmetic operations on the rearranged data objects from the input interface 1460. The image processor 1462 can: interpolate between two data objects with a

provided interpolation factor; multiply two data objects and divide the product by 255; multiply and add two data objects in general; round off fraction parts of a data object which may have various resolutions; clamp overflow of a data object to some maximum value and underflow of a data object to some minimum value; and perform scaling and clamping on a data object. The control signals on bus 1471 control which of the above arithmetic operations are performed on the data objects, and the order of the operations.

A ROM 1475 contains the dividends of $255/x$, where x is from 0 to 255, rounded in 8.8 format. The ROM 1475 is connected to the input interface 1460 and the image data processor 1462 via bus 1476. The ROM 1475 is used to generate blends of short lengths and multiply one data object by 255 and dividing the product by another data object.

Preferably, the number of operand buses eg 1452 is limited to 2, which is sufficient for most image processing operations.

Fig. 130 illustrates the input interface 1460 in further detail. Input interface 1460 includes data object interface unit 1480, operand interface units 1482 and 1484, address generation state machine 1486, blend generation state machine 1488, matrix multiplication state machine 1490, interpolation state machine 1490, data synchronizer 1500, arithmetic unit 1496, miscellaneous register 1498, and data distribution logic 1505.

Data object interface unit 1480 and operand interface units 1482 and 1484 are responsible to receive data objects and operands from outside. These interface units 1482, 1484 are all configured by control signals from control bus 1515. These interface units 1482, 1484 have data registers within them to contain the data objects/operands that they have just received, and they all produce a VALID signal which is asserted when the data within the data register is valid. The outputs of the data registers in these interface units 1482, 1484 are connected to data bus 1505. The VALID signals of these interface units 1482, 1484 are connected to flow bus 1510. When configured to fetch operands, operand interface units 1482 and 1484 accept

addresses from arithmetic unit 1496, matrix multiplication state machine 1490 and/or the output of data register in data object interface unit 1480, and select amongst them the required address in accordance with the control signals from control bus 1515. In some cases, the data registers in operand
 5 interface units 1482 and 1484 can be configured to store data from the output of data register in data object interface unit 1480 or arithmetic unit 1496, especially when they are not needed to accept and store data from outside.

Address generation state machine 1486 is responsible for controlling arithmetic unit 1496 so that it calculates the next coordinates to be accessed
 10 in the source image in affine image transformation operations and convolution operations.

The address generation state machine 1486 waits for START signal on control bus 1515 to be set. When the START signal on control bus 1515 is set, address generation state machine 1486 then de-asserts the STALL signal
 15 to data object interface unit 1480, and waits for data objects to arrive. It also sets a counter to be the number of data objects in a kernel descriptor that address generation state machine 1486 needs to fetch. The output of the counter is decoded to become enable signals for data registers in operand interface units 1482 and 1484 and miscellaneous register 1498. When the
 20 VALID signal from data object interface unit 1480 is asserted, address generation state machine 1486 decrements the counter, so the next piece of data object is latched into a different register.

When the counter reaches zero, address generation state machine 1486 tells operand interface unit 1482 to start fetching index table values and
 25 pixels from operand interface unit 1484. Also, it loads two counters, one with the number of rows, another with the number of columns. At every clock edge, when it is not paused by STALL signals from the operand interface unit 1482 or others, the counters are decremented to give the remaining rows and columns, and the arithmetic unit 1496 calculates the next coordinates to be
 30 fetched from. When both counters have reached zero, the counters reload

themselves with the number of rows and columns again, and arithmetic unit 1496 is configured to find the top left hand corner of the next matrix.

If interpolation is used to determine the true value of a pixel, address generation state machine 1486 decrements the number of rows and columns after every second clock cycle. This is implemented using a 1-bit counter, with the output used as the enable of the row and column counter. After the matrix is traversed around once, the state machine sends a signal to decrement the count in the length counter. When the counter reaches 1, and the final index table address is sent to the operand interface unit 1482, the state machine asserts a final signal, and resets the start bit.

Blend generation state machine 1488 is responsible for controlling arithmetic unit 1496 to generate a sequence of numbers from 0 to 255 for the length of a blend. This sequence of numbers is then used as the interpolation factor to interpolate between the blend start value and blend end value.

Blend generation state machine 1488 determines which mode it should run in (jump mode or step mode). If the blend length is less than or equal to 256, then jump mode is used, otherwise step mode is used.

The blend generation state machine 1488 calculates the following and puts them in registers (reg0, reg1, reg2). If a blend ramp is in step mode for a predetermined length, then latch 511-length in reg0 (24 bits), $512 - 2 * \text{length}$ in reg 1 (24 bits), and end-start in reg 2 (4x9 bits). If the ramp is in jump mode, then latch 0 into reg0, $255/(\text{length}-1)$ into reg1, and end-start into reg2 (4x9 bits).

In step mode, the following operations are performed for every cycle:

If $\text{reg0} > 0$, then add reg0 with reg 1 and store the result in reg0. Another incrementor can also be enabled so its output is incremented by 1. If $\text{reg0} \leq 0$, then add reg0 with 510 and store the result in reg0. Incrementor is not incremented. The output of the incrementor is the ramp value.

In jump mode, the following is done for every cycle:

Add reg0 with reg1. The Adder output is 24 bits, in fixed point format of 16.8. Store the adder output in reg0. If the first bit of fraction result is 1, then increment the integer part.

The least 8 bits of the integer part of the incrementor is the ramp value. The ramp value, the output of reg2, and the blend start value is then fed into the image data processor 1462 to produce the ramp.

Matrix multiplication state machine 1490 is responsible for performing linear color space conversion on input data objects using a conversion matrix. The conversion matrix is of the dimension 4X5. The first four columns multiply with the 4 channels in the data object, while the last column contains constant coefficients to be added to the sum of products. When the START signal from control bus 1515 is asserted, matrix multiplication state machine does the following:

- 1) It generates line numbers to fetch constant coefficients of the conversion matrix from buses 1482 and 1484. It also enables miscellaneous register 1498 to store these constant coefficients.

- 2) It contains a 1-bit flipflop, which generates a line number which is used as an address to fetch half of matrix from buses 1482 and 1484. It also generates a "MAT_SEL" signal that selects which half of the data object to be multiplied with that half of matrix.

- 3) It finishes when there is no data objects coming from data object interface unit 1480.

Interpolation state machine 1494 is responsible for performing horizontal interpolation of data objects. During horizontal interpolation, main data path unit 242 accepts a stream of data objects from bus 1451, and interpolates between adjacent data objects to output a stream of data objects which is twice or 4 times as long as the original stream. Since the data objects can be packed bytes or pixels, interpolation state machine 1494 operates differently in each case to maximize the throughput. Interpolation state machine 1494 does the following:

1) It generates INT_SEL signal to data distribution logic 1503 to rearrange the incoming data objects so that the right pair of data objects are interpolated.

2) It generates interpolation factors to interpolate between adjacent pairs of data objects.

3) It generates a STALL signal to stop data object interface unit 1480 from accepting more data objects. This is necessary as the output stream is longer than the input stream. The STALL signal goes to flow bus 1510.

Arithmetic unit 1496 contains circuitry for performing arithmetic calculations. It is configured by control signals on control bus 1515. It is used by two instructions only: affine image transformation and convolution, and blend generation in compositing.

In affine image transformation and convolution, arithmetic unit 1496 is responsible for:

1) Calculating the next x and y coordinates. To calculate x coordinates arithmetic unit 1496 uses an adder/subtractor to add/subtract the x part of horizontal and vertical delta to/from the current x coordinate. To calculate the y coordinates arithmetic unit 1498 uses an adder/subtractor to add/subtract the y part of the horizontal or vertical delta to/from the current y coordinate.

2) Adding the y coordinate to the index table offset to calculate the index table address. This sum is also incremented by 4 to find the next index table entry, when interpolation is used to find true value of a pixel.

3) Adding the x coordinate to the index table entry to find the address of the pixel.

4) Subtract 1 from the length count.

In blend generation, arithmetic unit 1496 does the following:

1) In step mode, one of the ramp adders is used to calculate an internal variable in the ramp generation algorithm, while the other adder is

used to increment the ramp value when the internal variable is greater than 0.

2) In jump mode, only one of the adders is required to add the jump value to the current ramp value.

5 3) Round off fractions occur in jump mode.

4) Subtract start of blend from end of blend at the beginning of ramp generation.

5) Subtract one from the length count.

Miscellaneous register 1498 provides extra storage space apart from
10 the data registers in data object interface unit 1480 and operand interface units 1482 and 1484. It is usually used to store internal variables or as a buffer of past data objects from data object interface unit 1480. It is configured by control signals on control bus 1515.

Data synchronizer 1500 is configured by control signals on control bus
15 1515. It provides STALL signals to data object interface unit 1480 and operand interface units 1482 and 1484 so that if one of the interface units receives a piece of data object others have not, that interface unit is stalled until all the other interface units have received their pieces of data.

Data distribution logic 1505 rearranges data objects from data bus
20 1510 and register file 1472 via bus 1530 in accordance with control signals on control bus 1515, including a MAT_SEL signal from matrix multiplication state machine 1490 and a INT_SEL signal from interpolation state machine 1494. The rearranged data is outputted onto bus 1461.

Fig. 131 illustrates image data processor 1462 of Fig. 129 in further
25 detail. Image data processor 1462 includes a pipeline controller 1540, and a number of color channel processors 1545, 1550, 1555 and 1560. All color channel processors accept inputs from bus 1565, which is driven by the input interface 1460 (Fig. 131). All color channel processors and pipeline controller 1540 are configured by control signals from control signal register 1470 via
30 bus 1472. All the color channel processors also accept inputs from register

file 1472 and ROM 1475 of Fig. 129 via bus 1580. The outputs of all the color channel processors and pipeline controller are grouped together to form bus 1570, which forms the output 1455 of image data processor 1462.

Pipeline controller 1540 controls the flow of data objects within all the color channel processors by enabling and disabling registers within all the color channel processors. Within pipeline controller 1540 there is a pipeline of registers. The shape and depth of the pipeline is configured by the control signals from bus 1471, and the pipeline in pipeline controller 1540 has the same shape as the pipeline in the color channel processors. The Pipeline controller accepts VALID signals from bus 1565. For each pipeline stage within pipeline controller 1540, if the incoming VALID signal is asserted and the pipeline stage is not stalled, then the pipeline stage asserts the register enable signals to all color channel processors, and latch the incoming VALID signal. The output of the latch then a VALID signal going to the next pipeline stage. In this way the movement of data objects in the pipeline is simulated and controlled, without storage of any data.

Color channel processors 1545, 1550, 1555 and 1560 perform the main arithmetic operations on incoming data objects, with each of them responsible for one of the channels of the output data object. In the preferred embodiment the number of color channel processors is limited to 4, since most pixel data objects have a maximum of 4 channels.

One of the color channel processors processes the opacity channel of a pixel. There is additional circuitry (not shown in Fig. 131), connected to the control bus 1471, which transforms the control signals from the control bus 1471 so that the color channel processor processes the opacity channel correctly, as for some image processing operations the operations on the opacity channel is slightly different from the operations on the color channels.

Fig. 132 illustrates color channel processor 1545, 1550, 1555 or 1560 (generally denoted by 1600 in Fig. 132) in further detail. Each color channel processor 1545, 1550, 1555 or 1560 includes processing block A 1610,

processing block B 1615, big adder 1620, fraction rounder 1625, clamp-or-wrapper 1630, and output multiplexer 1635. The color channel processor 1600 accepts control signals from control signal register 1470 via bus 1602, enable signals from pipeline controller 1540 via bus 1604, information from
 5 register file 1472 via bus 1605, data objects from other color channel processor via bus 1603, and data objects from input interface 1460 via bus 1601.

Processing block A 1610 performs some arithmetic operations on the data objects from bus 1601, and produces partially computed data objects on
 10 bus 1611. The following illustrates what processing block A 1610 does for designated image processing operations.

In compositing, processing block A 1610 pre-multiplies data objects from data object bus 1451 with opacity, interpolates between a blend start value and a blend end value with an interpolation factor from input interface
 15 1460 in Fig. 129, pre-multiplies operands from operand bus 1452 in Fig. 129 or multiplies blend color by opacity, and attenuates multiplication on pre-multiplied operand or blend color data.

In general color space conversion, the processing block A 1610 interpolates between 4 color table values using two fraction values from bus
 20 1451 in Fig. 129.

In affine image transformation and convolution, the processing block A 1610 pre-multiplies the color of the source pixel by opacity, and interpolates between pixels on the same row using the fraction part of current x-coordinate.

In linear color space conversion, the processing block A 1610 pre-
 25 multiplies color of the source pixel by opacity, and multiplies pre-multiplied color data with conversion matrix coefficients.

In horizontal interpolation and vertical interpolation, the processing block A 1610 interpolates between two data objects.

In residual merging, the processing block A 1610 adds two data
 30 objects.

Processing block A 1610 includes a number of multifunction blocks 1640 and processing block A glue logic 1645. The multifunction blocks 1640 are configured by control signals, and may perform any one of the following functions:

- 5 add/subtract two data objects;
- passing one data object;
- interpolate between two data objects with a interpolation factor;
- pre-multiply a color with an opacity;
- multiply two data objects, and then add a third data object to the
- 10 product; and
- add/subtract two data objects, and then pre-multiply the
- sum/difference with an opacity.

The registers within the multifunction blocks 1640 are enabled or disabled by enable signals from bus 1604 generated by pipelined controller 1540 in Fig. 131. Processing block A glue logic 1645 accepts data objects from bus 1601 and data objects from bus 1603, and the outputs of some of the multifunction blocks 1640, and routes them to inputs of other selected multifunction blocks 1640. Processing block A glue logic 1645 is also configured by control signals from bus 1602.

20 Processing block B 1615 performs arithmetic operations on the data objects from bus 1601, and partially computed data objects from bus 1611, to produce partially computed data objects on bus 1616. The following description illustrates what processing block B 1615 does for designated image processing operations.

25 In compositing (with non-plus operators), the processing block B 1615 multiplies pre-processed data objects from data object bus 1451 and operands from operand bus 1452 with compositing multiplicands from bus 1603, and multiplies clamped/wrapped data objects by output of the ROM, which is 255/opacity in 8.8 format.

7/8
In compositing with plus operator, the processing block B 1615 adds two pre-processed data objects. In the opacity channel, it also subtracts 255 from the sum, multiplies an offset with the difference, and divides the product by 255.

5 In general color space conversion, the processing block B 1615 interpolates between 4 color table values using 2 of the fraction values from bus 1451, and interpolates between partially interpolated color value from processing block A 1610 and the result of the previous interpolation using the remaining fraction value.

10 In affine image transformation and convolution, the processing block B 1615 interpolates between partially interpolated pixels using the fraction part of current y-coordinate, and multiplies interpolated pixels with coefficients in a sub-sample weight matrix.

In linear color space conversion, the processing block B 1615 pre-
15 multiplies the color of the source pixel by opacity, and multiplies pre-multiplied color with conversion matrix coefficients.

Processing block B 1615 again includes a number of multifunction blocks and processing block B glue logic 1650. The multifunction blocks are exactly the same as those in processing block A 1610, but the processing block
20 B glue logic 1650 accepts data objects from buses 1601, 1603, 1611, 1631 and the outputs of selected multifunction blocks and routes them to the inputs of selected multifunction blocks. Processing block B glue logic 1650 is also configured by control signals from bus 1602.

Big adder 1620 is responsible for combining some of the partial
25 results from processing block A 1610 and processing block B 1615. It accepts inputs from input interface 1460 via bus 1601, processing block A 1610 via bus 1611, processing block B 1615 via bus 1616, and register file 1472 via bus 1605, and it produces the combined result on bus 1621. It is also configured by control signals on bus 1602.

For various image processing operations, big adder 1620 may be configured differently. The following description illustrates its operation during designated image processing operations.

In compositing with non-plus operators, the big adder 1620 adds two
5 partial products from processing block B 1615 together.

In compositing with plus operator, the big adder 1620 subtracts the sum of pre-processed data objects with offset from the opacity channel, if an offset enable is on.

In affine image transformation/convolution, the big adder 1620
10 accumulates the products from processing block B 1615

In linear color space conversion, in the first cycle, the big adder adds the two matrix coefficients/data object products and the constant coefficient together. In the second cycle, it adds the sum of last cycle with another two matrix coefficients/data object products together.

15 Fraction rounder 1625 accepts input from the big adder 1620 via bus 1621 and rounds off the fraction part of the output. The number of bits representing the fraction part is described by a BP signal on bus 1605 from register file 1472. The following table shows how the BP signal is interpreted. The rounded output is provided on bus 1626.

20

Table 27: Fraction Table

bp field	Meaning
0	Bottom 26 bits are fractions.
1	Bottom 24 bits are fractions.
2	Bottom 22 bits are fractions.
3	Bottom 20 bits are fractions.
4	Bottom 18 bits are fractions.
5	Bottom 16 bits are fractions.
6	Bottom 14 bits are fractions.
7	Bottom 12 bits are fractions.

As well as rounding off fraction, fraction rounder 1625 also does two things:

- 1) determines whether the rounded result is negative; and
- 2) determines whether the absolute value of the rounded result is greater than 255.

Clamp-or-wrapper 1630 accepts inputs from fraction rounder 1625 via
 5 bus 1626 and does the following in the order described:

finds the absolute value of the rounded result, if such option is enabled; and

clamps any underflow of the data object to the minimum value of the data object, and any overflow of the data object to the maximum value of the
 10 data object.

Output multiplexer 1635 selects the final output from the output of processing block B on bus 1616 and the output of clamp-or-wrapper on bus 1631. It also performs some final processing on the data object. The following description illustrates its operation for designated image processing
 15 operations.

In compositing with non-plus operators and un-pre-multiplication, the multiplexer 1635 combines some of the outputs of processing block B 1615 to form the un-pre-multiplied data object.

In compositing with non-plus operator and no un-pre-multiplication,
 20 the multiplexer 1635 passes on the output of clamp-or-wrapper 1630.

In compositing with plus operator, the multiplexer 1635 combines some of the outputs of processing block B 1630 to form resultant data object.

In general color space conversion, the multiplexer 1635 applies the translate-and-clamp function on the output data object.

25 In other operations, the multiplexer 1635 passes on the output of clamp-or-wrapper 1630.

Fig. 133 illustrates a single multifunction block (e.g. 1640) in further detail. Multifunction block 1640 includes mode detector 1710, two addition operand logic units 1660 and 1670, 3 multiplexing logic units 1680, 1685 and

1690, a 2-input adder 1675, a 2-input multiplier with 2 addends 1695, and register 1705.

Mode detector 1710 accepts one input from control signal register 1470, in Fig. 129 the MODE signal 1711, and two inputs from input interface 1460, in Fig. 129 SUB signal 1712 and SWAP signal 1713. Mode detector 1710 decodes these signals into control signals going to addition operand logic units 1660 and 1670, and multiplexing logic units 1680, 1685 and 1690, and these control signals configure multifunction block 1640 to perform various operations. There are 8 modes in multifunction block 1640:

1) Add/sub mode: adds or subtract input 1655 to/from input 1665, in accordance with the SUB signal 1712. Also, the inputs can be swapped in accordance with the SWAP signal 1713.

2) Bypass mode: bypass input 1655 to output.

3) Interpolate mode: interpolates between inputs 1655 and 1665 using input 1675 as the interpolation factor. Inputs 1655 and 1665 can be swapped in accordance with the SWAP signal 1713.

4) Pre-multiply mode: multiplies input 1655 with input 1675 and divide it by 255. The output of the INC register 1708 tells the next stage whether to increment the result of this stage in bus 1707 to obtain the correct result.

5) Multiply mode: multiplies input 1655 with 1675.

6) Add/subtract-and-pre-multiply mode: adds/subtracts input 1665 to/from input 1655, multiplies the sum/difference with input 1675, and then divide the product by 255. The output of the INC register 1708 tells the next stage whether to increment the result of this stage in bus 1707 to obtain the correct result.

Addition operand logic units 1660 and 1670 find one's complement of the input on demand, so that the adder can do subtraction as well. Adder 1675 adds the outputs of addition operand logic 1660 and 1670 in buses 1662 and 1672 together, and outputs the sum in bus 1677.

Multiplexing logic 1680, 1685 and 1690 select suitable multiplicands and addends to implement, a desired function. They are all configured by control signals on bus 1714 from mode detector 1710.

Multiplier with two addends 1695 multiplies input from bus 1677
5 with input from bus 1682, then adds the products to the sum of inputs from buses 1687 and 1692.

Adder 1700 adds the least significant 8 bits of the output of multiplier 1695 with the most significant 8 bits of the output of multiplier 1695. The carryout of adder 1700 is latched in INC register 1701. INC
10 register 1701 is enabled by signal 1702. Register 1705 stores the product from multiplier 1695. It is also enabled by signal 1702.

Fig. 134 illustrates a block diagram for the compositing operations. The compositing operation accepts three input streams of data:

- 1) The accumulated pixel data, which is derived from the same
15 location as the result is stored to in this accumulator model.
- 2) A compositing operand - which consists of color and opacity. The color and opacity can both be either flat, a blend, pixels or tiled.
- 3) Attenuation - which attenuates the operand data. The attenuation can be flat, a bit map or a byte map.

Pixel data typically consists of four channels. Three of these
20 channels make up the color of the pixel. The remaining channel is the opacity of the pixel. Pixel data can be pre-multiplied or normal. When pixel data is pre-multiplied, each of the color channels are multiplied with the opacity. Since equations for compositing operators are simple with pre-multiplied
25 pixels, usually pixel data is pre-multiplied before it is composited with another pixel.

The compositing operators implemented in the preferred embodiments are shown in Table 1. Each operator works on pre-multiplied data. (a_{co}, a_o) refers to a pre-multiplied pixel of color a_c and opacity a_o , r is
30 the "offset" value and $wc()$ is the wrapping/clamping operator the reverse

operator of each of the over, in, out, atop operators in Table 1 is also implemented, and the compositing model has the accumulator on the left.

Composite block 1760 in Fig. 134 comprizes three color sub-blocks and a opacity sub-block. Each color sub-block operates on one color channel, and opacity channel of the input pixels to obtain the color of the output pixel. The following pseudo code shows how this is done.

```

    PIXEL Composite(          IN colorA, colorB: PIXEL;
                              IN opacityA, opacityB: PIXEL;
10      IN                      comp_op:
COMPOSITE_OPERATOR
    )
(
    PIXEL result;
15    IF comp_op is rover, rin, rout, ratop THEN
        swap colorA and colorB;
        swap opacityA and opacityB;
    END IF;
    IF comp_op is over or rover or loado or plus THEN
20      X = 1;
    ELSE IF comp_op is in or rin or atop or ratop THEN
        X = opacityB;
    ELSE IF comp_op is out or rout or xor THEN
        X = not(opacityB);
    ELSE IF comp_op is loadzero or loadc or loadco THEN
25      X = 0
    END IF;
    IF comp_op is over or rover or atop or ratop or xor THEN
        Y = not(opacityA);
30    ELSE IF comp_op is plus or loadc or loadco THEN

```

```

        Y = not(opacityA);
    ELSE IF comp_op is plus or loadc or loadco THEN
        Y = 1;
    ELSE IF comp_op is in or rin or out or rout or
5      loadzero or loado THEN
        Y = 0
    END IF;
    result = colorA * X + colorB * Y;
    RETURN result;

```

10

The above pseudo code is different for the opacity sub-block, since the operators 'loadc' and 'loado' have different meaning in the opacity channel.

Block 1765 in Fig. 134 is responsible for clamping or wrapping the output of block 1760. When block 1765 is configured to clamp, it forces all
 15 values less than the minimum allowable value to the minimum allowed value, and all values more than the maximum allowed value to the maximum allowed value. If block 1765 is configured to wrap, it calculates the following equation:

$$((x - \min) \bmod (\max - \min)) + \min,$$

20 whereby min and max are the minimum and maximum allowed value of the color respectively. Preferably the minimum value for a color is 0, and the maximum value is 255.

Block 1770 in Fig. 134 is responsible for un-pre-multiplying the result from block 1765. It un-pre-multiplies a pixel by multiplying the pre-
 25 multiplied color value with $255/o$, where o is the opacity after composition. The value $255/o$ is obtained from a ROM inside the compositing engine. The value stored in the ROM is in the format of 8.8 and the rest of the fraction is rounded. The result of multiplication is stored in the format of 16.8. The result would be rounded to 8 bits to produce the un-pre-multiplied pixel.

Blend generator 1721 generates a blend of a specified length with specified start and end values. Blend generation is done in two stages:

- 1) ramp generation, and
- 2) interpolation

5 In ramp generation, the compositing engine generates a linearly increasing number sequence from 0 to 255 over the length of the instruction. There are two modes in ramp generation: the "jump" mode, when the length is less than or equal to 255, and the "step" mode when the length is greater than 255. The mode is determined by examining the 24 most significant bits of the length. In the jump mode, the ramp value increases by at least one in every clock period. In the step mode, the ramp value increases by at most one in every clock period.

10 In the jump mode, the compositing engine uses the ROM to find out the step value $255/(\text{length}-1)$, in 8.8 format. This value is then added to a 16-bit accumulator. The output of the accumulator is rounded to 8 bits to form the number sequence. In the step mode, the compositing engine uses an algorithm similar to Bresenham's line drawing algorithm, as described by the following pseudo code.

```
20 Void linedraw (    length: INTEGER
                    )
```

```
{
```

```
    d = 511 - length;
```

```
    incrE = 510;
```

```
    incrNE = 512 - 2*length;
```

```
    ramp = 0;
```

```
    for (i=0; i(length; i++)
```

```
    {
```

```
        if d (>= 0 then
```

```
            d += incrE;
```

```
        else {
```

```

    d += incrNE;
    ramp++;
  }
}
5  }

```

After that, the following equation is calculated to generate the blend from the ramp.

$$\text{Blend} = ((\text{end} - \text{start}) \times \text{ramp} / 255) + \text{start}$$

The division by 255 is rounded. The above equation requires 2 adders and a block that "pre-multiplies" (end-start) by ramp for each channel.

Another image processing operation that the main data path unit 242 is able to perform is general color space conversion. Generalized Color Space Conversion (GCSC) uses piecewise tri-linear interpolation to find out the output color value. Preferably, conversion is from a three dimensional input space to one or four dimensional output space.

In some cases, there is a problem with the accuracy of tri-linear interpolation at the edges of the color gamut. This problem is most noticeable in printing devices that have high sensitivity near an edge of the gamut. To overcome this problem, GCSC can optionally be calculated in an expanded output color space and then scaled and clamped to the appropriate range using the formula in equation:

$$\text{out} = \begin{array}{ll} 0 & \text{if } x < 63 \\ 2(x-64) & \text{if } (64 \leq x < 191) \\ 255 & \text{if } (192 \leq x) \end{array}$$

Yet other image processing operations that the preferred embodiment is able to perform are image transformation and convolution. In image transformation, the source image is scaled, rotated, or skewed to form the destination image. In convolution, the source image pixels are sampled with

a convolution matrix to provide the destination image. To construct a scanline in the destination image, the following steps are required:

- 1) Perform an inverse transform of the scanline in the destination image back to the source image as illustrated in Fig. 135. This tells what pixels in the source image are needed to construct that scanline in the destination image.
- 2) Decompress the necessary portions of the source image.
- 3) Inverse-transform the starting x and y coordinates, horizontal and vertical subsampling distances in the destination image back to source image.
- 4) Pass all these information to the processing units which performs the necessary sub-sampling and/or interpolation to construct the output image pixel by pixel.

The calculations to work out which parts of the source image are relevant, sub-sampling frequencies to use, etc, are performed by the host application. Sub-sampling, interpolation, and writing the pixels into the destination image memory are done by the preferred embodiments.

Fig. 136 shows a block diagram of the steps required to calculate the value for a destination pixel. In general, the computation-intensive part is the bi-linear interpolation. The block diagram in Fig. 136 assumes that all the necessary source image pixels are available.

The final step in calculating a destination pixel is to add together all the possibly bi-linearly interpolated sub-samples from the source image. These values are given different weights.

Fig. 137 illustrates a block diagram of the image transformation engine that can be derived from suitable settings within the main data path unit 242. Image transformation engine 1830 includes address generator 1831, pre-multiplier 1832, interpolator 1833, accumulator 1834, and logic for rounding, clamping and finding absolute value 1835.

Address generator 1831 is responsible for generating x and y coordinates of the source image which are needed to construct a destination

pixel. It also generates addresses to obtain index offsets from an input index table 1815 and pixels from image 1810. Before address generator 1831 begins generating x and y coordinates in the source image, it reads in a kernel descriptor. These are two formats of kernel descriptors. They are shown in Fig. 138. The kernel descriptor describes:

- 1) Source image start coordinates (unsigned fixed point, 24.24 resolution). Location (0,0) is at the top left of the image.
- 2) Horizontal and vertical sub-sample deltas (2's complement fixed point, 24.24 resolution).
- 10 3) a 3 bit **bp** field defining the location of the binary point within the fixed point matrix coefficients. The definition and interpretation of the **bp** field is shown in Fig. 150.
- 4) Accumulation matrix coefficients. These are of "variable" point resolution of 20 binary places (2's complement), with the location of the binary point implicitly specified by the **bp** field.
- 15 5) an **rl** field that indicates the remaining number of words in the kernel descriptor. This value is equal to the number of rows times the number of columns minus 1.

For the short kernel descriptor, apart from the integer part of start x coordinate, the other parameters are assumed to have the following values:

starting x coordinate fraction <-0,

starting y coordinate <-0,

horizontal delta <-1.0,

vertical delta <-1.0.

25 After address generator 1831 is configured, it calculates the current coordinates. It does this in two different ways, depending on the dimensions of the subsample matrix. If the dimensions of the subsample matrix are 1 x 1, address generator 1831 adds the horizontal delta to the current coordinates until it has generated enough coordinates.

If the dimensions of the subsample matrix are not 1×1 , address generator 1831 adds the horizontal delta to the current coordinates until one row of the matrix is finished. After that, address generator 1831 adds the vertical delta to the current coordinates to find the coordinates on the next row. After that, address generator 1831 subtracts the horizontal delta from the current coordinates to find the next coordinates, until one more row is finished. After that, address generator 1831 adds the vertical delta to the current coordinates and the procedure is repeated again. Top diagram in Fig. 150 illustrates this method of accessing the matrix. Using this scheme, the matrix is traversed in a zig-zag way, and fewer registers are required since the current x and y coordinates are calculated using the above method, the accumulation matrix coefficients must be listed in the kernel descriptor in the same order.

After generating the current coordinates, the address generator 1831 adds the y coordinate to the index table base address to get the address to the index table. (In case when source pixels are interpolated, address generator 1831 needs to obtain the next index table entry as well.) The index table base address should point to the index table entry for $y + 0$. After obtaining the index offset from the index table, the address generator 1831 adds that to the x coordinate. The sum is used to get 1 pixel from the source image (or 2 if source pixels are interpolated). In case when source pixels are interpolated, the address generator 1831 adds the x coordinates to the next index offset, and two more pixels are obtained.

Convolution uses a similar method to generate coordinates to image transformation. The only difference is that in convolution, the start coordinates of the matrix for the next output pixel is one horizontal delta away from the starting coordinates of the matrix for the previous pixel. In image transformation, the starting coordinates of the matrix for the next pixel is one horizontal delta away from the coordinates of the top right pixel in the matrix for the previous output pixel.

The middle diagrams in Fig. 139 illustrates this difference.

Pre-multiplier 1832 multiplies the color channels with the opacity channel of the pixel if required.

Interpolator 1832 interpolates between source pixels to find the true color of the pixel required. It gets two pixels from the source image memory at all times. Then it interpolates between those two pixels using the fraction part of the current x coordinate and puts the result in a register. After that, it obtains the two pixels on the next row from the source image memory. Then it interpolates between those two pixels using the same x fraction. After that, interpolator 1833 uses the fraction part of the current y coordinate to interpolate between this interpolated result and the last interpolated result.

Accumulator 1834 does two things:

- 1) it multiplies the matrix coefficients with the pixel, and
- 2) it accumulates the product above until the whole matrix is traversed. Then it outputs a value to the next stage.

Preferably the accumulator 1834 can be initialized with 0 or a special value on a channel-by-channel basis.

Block 1835 rounds the output of accumulator 1834, then clamps any underflows or overflows to the maximum and minimum values if required, and finds the absolute value of the output if required. The location of the binary point within the output of the accumulator is specified by the *bp* field in the kernel descriptor. The *bp* field indicates the number of leading bits in the accumulated result to discard. This is shown in the bottom diagram of Fig. 139. Note that the accumulated value is treated as a signed two's complement number.

Yet another image processing operation that the main data path unit 242 can perform is matrix multiplication. Matrix Multiplication is used for color space conversion where an affine relationship exists between the two

spaces. This is distinct from General Color Space Conversion (based on tri-linear interpolation).

The result of Matrix Multiplication is defined by the following equation:

5

$$\begin{bmatrix} r_x \\ r_y \\ r_z \\ r_0 \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} & b_{0,4} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} & b_{1,4} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} & b_{2,4} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} & b_{3,4} \end{bmatrix} \begin{bmatrix} a_x \\ a_y \\ a_z \\ a_0 \\ 255 \end{bmatrix}$$

where r_i is the result pixel and a_i is the A operand pixel. Matrix must be 5 columns by 4 rows.

10

Fig. 140 illustrates a block diagram of the multiplier-adders that perform the matrix multiplication in the main data path unit 242. It includes multipliers to multiply the matrix coefficients with the pixel channels, adders to add the products together, and logic to clamp and find the absolute value of the output if required.

15

The complete matrix multiplication takes 2 clock cycles to complete. At each cycle the multiplexers are configured differently to select the right data for the multipliers and adders.

At cycle 0, the least significant 2 bytes of the pixel are selected by the multiplexers 1851, 1852. They then multiply the coefficients on the left 2 columns of the matrix, i.e. the matrix coefficients on line 0 in the cache. The results of the multiplication, and the constant term in the matrix, are then added together and stored.

At cycle 1, the more significant 2 bytes of the pixel are selected by the top multiplexers. They then multiply the coefficients on the right 2 columns of the matrix.

25

The result of the multiplication is then added 1854 to the result of the last cycle. The sum of the adder is then rounded 1855 to 8 bits.

The 'operand logic' 1856 rearranges the outputs of the multipliers to form four of the inputs of the adder 1854. It rearranges the outputs of the multipliers so that they can be added together to form the true product of the 24-bit coefficient and 8-bit pixel component.

5 The 'AC (Absolute value-clamp/wrap) logic' 1855 firstly rounds off the bottom 12 bits of the adder output. It then finds the absolute value of the rounded result if it is set to do so. After that it clamps or wraps the result according to how it is set up. If the 'AC logic' is set to clamp, it forces all values less than 0 to 0 and all values more than 255 to 255. If the 'AC logic' is set to wrap, the lower 8 bits of the integer part is passed to the output.

10 Apart from the image processing operations above, the main data path unit 242 can be configured to perform other operations.

The foregoing description provides a computer architecture that is capable of performing various image processing operations at high speed, while the cost is reduced by design reuse. The computer architecture described is also highly flexible, allowing any external programming agent with intimate knowledge of the architecture to configure it to perform image processing operations that were not initially expected. Also, as the core of the design mainly comprizes a number of those multifunction blocks, the design effort is reduced significantly.

3.18.6 Data Cache Controller and Cache

25 The data cache controller 240 maintains a four-kilobyte read data cache 230 within the coprocessor 224. The data cache 230 is arranged as a direct mapped RAM cache, where any one of a group of lines of the same length in external memory can be mapped directly to the same line of the same length in cache memory 230 (Fig 2). This line in cache memory is commonly referred to as a cache-line. The cache memory comprizes a multiple number of such cache-lines.

30 The data cache controller 240 services data requests from the two operand organizers 247, 248. It first checks to see if the data is resident in

cache 230. If not, data will be fetched from external memory. The data cache controller 240 has a programmable address generator, which enables the data cache controller 240 to operate in a number of different addressing modes. There are also special addressing modes where the address of the data requested is generated by the data cache controller 240. The modes can also involve supplying up to eight words (256 bits) of data to the operand organizers 247, 248 simultaneously.

The cache RAM is organized as 8 separately addressable memory banks. This is needed for some of the special addressing modes where data from each bank (which is addressed by a different line address) is retrieved and packed into 256 bits. This arrangement also allows up to eight 32-bits requests to be serviced simultaneously if they come from different banks.

The cache operates in the following modes, which will be discussed in more detail later. Preferably, it is possible to automatically fill the entire cache if this is desired.

1. Normal Mode
2. Single Output General Color Space Conversion Mode
3. Multiple Output General Color Space Conversion Mode
4. JPEG Encoding Mode
5. Slow JPEG Decoding Mode
6. Matrix Multiplication Mode
7. Disabled Mode
8. Invalidate Mode

Fig. 141 shows the address, data and control flow of the data cache controller 240 and data cache 230 shown in figure 2.

The data cache 230, consists of a direct mapped cache of the type previously discussed. The data cache controller 240, consists of a tag memory 1872 having a tag entry for each cache-line, which tag entry comprizes the most significant part of the external memory address that the cache-line is currently mapped to. There is also a line valid status memory 1873 to

indicate whether the current cache-line is valid. All cache-lines are initially invalid.

The data cache controller 240 can service data requests from operand organizer B 247 (Fig 2) and operand organizer C 248 (Fig. 2) simultaneously via the operand bus interface 1875. In operation, one or both of the operand organizers 247 or 248 (Fig. 2), supplies an index 1874 and asserts a data request signal 1876. The address generator 1881 generates one or more complete external addresses 1877 in response to the index 1874. A cache controller 1878 determines if the requested data is present in cache 230 by checking the tag memory 1872 entries for the tag addresses of the generated addresses 1877 and checking the line valid status memory 1873 for the validity of the relevant cache-line(s). If the requested data is present in cache memory 230, an acknowledgment signal 1879 is supplied to the relevant operand organizer 247 or 248 together with the requested data 1880. If the requested data is not present in the cache 230, the requested data 1870 is fetched from external memory, via an input bus interface 1871 and the input interface switch 252 (Fig. 2). The data 1870 is fetched by asserting a request signal 1882 and supplying the generated address(es) 1877 of the requested data 1870. An acknowledgement signal 1883 and the requested data 1870 are then sent to the cache controller 1878 and the cache memory 230 respectively. The relevant cache-line(s) of the cache memory 230 are then updated with the new data 1870. The tag addresses of the new cache-line(s) are also written into tag memory 1872, and the line valid status 1873 for the new cache-line(s) are asserted. An acknowledgment signal 1879 is then sent to the relevant operand organizer 247 or 248 (Fig 2) together with the data 1870.

Turning now to Fig. 142, which shows the memory organization of the data cache 230. The data cache 230 is arranged as a direct mapped cache with 128 cache-lines C0,...,C127 and a cache-line length of 32 bytes. The cache RAM consists of 8 separately addressable memory banks B0,...,B7,

each having 128 bank-lines of 32 bits, with each cache-line C_i consisting of the corresponding 8 bank-lines B_{0i}, \dots, B_{7i} of the 8 memory banks B_0, \dots, B_7 .

The composition of the generated complete external memory address is shown in Fig. 143. The generated address is a 32-bit word having a 20-bit tag address, a 7-bit line address, a 3-bit bank address and a 2-bit byte address. The 20-bit tag address is used for comparing the tag address with the tag stored in the tag memory 1872. The 7-bit line address is used for addressing the relevant cache-line in the cache memory 1870. The 3-bit bank address is used for addressing the relevant bank of the memory banks of the cache memory 1870. The 2-bit byte address is used for addressing the relevant byte in the 32-bit bank line.

Turning now to Fig. 144, which shows a block diagram of the data cache controller 240 and data cache 230 arrangement. In this arrangement, a 128 by 256 bit RAM makes up the cache memory 230, and as noted previously is organized as 8 separately addressable memory banks of 128 by 32 bits. This RAM has a common write enable port (write), a common write address port (write_addr) and a common write data port (write_data). The RAM also has a read enable port (read), eight read address ports (read_addr) and eight read data output ports (read_data). A write enable signal is generated by the cache controller block 1878 for supply to the common write enable port (write) for simultaneously enabling writing to all of the memory banks of the cache memory 230. When required, the data cache 230 is updated by one or more lines of data from external memory via the common write data port (write_data). A line of data is written utilizing the 8:1 multiplexer MUX supplying the line address to the write address port (write_addr). The 8:1 multiplexer MUX selects the line address from the generated external addresses under the control of the data cache controller (addr_select). A read enable signal is generated by the cache controller block 1878 for supply to the common read port (read) for simultaneously enabling reading of all the memory banks of cache memory 230. In this way, eight different bank-lines

of data can be simultaneously read from eight read data ports (read_data) in response to respective line addresses supplied on the eight read address ports (read_addr) of the memory banks of the cache memory 230.

Each bank of the cache memory 230 has its own programmable address
5 generator 1881. This allows eight different locations to be simultaneously accessed from the respective eight banks of memory. Each address generator 1881 has a dcc-mode input for setting the mode of operation of the address generator 1881, an index-packet input, a base-address input and an address output. The modes of operation of the programmable address generator
10 1881 include

(a) Random access mode where a signal on the dcc-mode input sets each address generator 1881 to the random access mode and complete external memory address(es) are supplied on the index-packet input(s) and outputted on the address output of one or more of the address generators
15 1881; and

(b) JPEG encoding and decoding, color space conversion, and matrix multiplication modes, where a signal on the dcc-mode input sets each address generator 1881 to the appropriate mode. In these modes, each address generator 1881 receives an index on the index-packet input and generates an
20 index address. The index addresses are then added to a fixed base address supplied on the base-address input resulting in a complete external memory address which is then outputted on the address output. Depending upon the mode of operation, the address generators are able to generate up to eight different complete external memory addresses.

25 The eight address generators 1881 consist of eight different combinational logic circuits each having as their inputs; a base-address, a dcc-mode and an index and each having a complete external memory address as an output.

A base-address register 1885 stores the current base address that is combined with the index packet and a dcc-mode register 1888 stores the current operational mode (dcc-mode) of the data cache controller 240.

The tag memory 1872 comprizes one block of 128 by 20 bit, multi- port RAM. This RAM has one write port (update-line-addr), one write enable port (write), eight read ports (read0line-addr,...,read7line-addr) and eight read output ports (tag0_data,...,tag7_data). This enables eight simultaneous lookups on the ports (read0line-addr,...,read7line-addr) by the eight address generators 1881 to determine, for each line address of the one or more generated memory addresses, the tag addresses currently stored for those lines. The current tag addresses for those lines are outputted on the ports (tag0-data,...tag7-data) to the tag comparator 1886. When required, a tag write signal is generated by the cache controller block 1878 for supply to the write port (write) of the tag memory 1872 to enable writing to the tag memory 1872 on the port (update-line-addr).

A 128-bit line valid memory 1873 contains the line valid status for each cache-line of the cache memory 230. This is 128 by 1 bit memory with one write port (update-line-addr), one write enable port (update), eight read ports (read0line-addr,...,read7line-addr) and eight read output ports (linevalid0,...,linevalid7). In a similar manner to the tag memory, this allows eight simultaneous lookups on the ports (read0line-addr,...,read7line-addr) by the eight address generators 1881 to determine, for each line address of the one or more generated memory addresses, the line valid status bits currently stored for those lines. The current line valid bits for those lines are outputted on the ports (linevalid0,...,linevalid7) to the tag comparator 1886. When required, a write signal is generated by the cache controller block 1878 for supply to the write port (update) of the line valid status memory 1873 to enable writing to the line valid status memory 1873 on the port (update-line-addr).

The tag comparator block 1886 consists of eight identical tag comparators having; tag_data inputs for respectively receiving the tag addresses currently stored in tag memory 1872 at those lines accessed by the line addresses of the currently generated complete external addresses, tag_addr inputs for respectively receiving the tag addresses of the currently generated complete external memory addresses, a dcc_input for receiving the current operational mode signal (dcc_mode) for setting the parts of the tag addresses to be compared, and a line_valid input for receiving the line valid status bits currently stored in the line valid status memory 1873 at those lines accessed by the line addresses of the currently generated complete external memory addresses. The comparator block 1886 has eight hit outputs for each of the eight address generators 1881. A hit signal is asserted when the tag address of the generated complete external memory address matches the contents of the tag memory 1872 at the location accessed by the line address of the generated complete external memory address, and the line valid status bit 1873 for that line is asserted. In this particular embodiment, the data structures stored in external memory are small, and hence the most significant bits of the tag addresses are the same. Thus it is preferable to compare only those least significant bits of the tag addresses which may vary. This is achieved by the current operational mode signal (dcc_mode) setting the tag comparator 1886 for comparing those least significant bits of the tag addresses which may vary.

The cache controller 1878 accepts a request (proc_req) 1876 from the operand B 247 or operand C 248 and acknowledges (proc_ack) 1879 this request if the data is available in cache memory 230. Depending on the mode of operation, up to eight differently addressed data items may be requested, one from each of the eight banks of cache memory 230. The requested data is available in cache memory 230 when the tag comparator 1886 asserts a hit for that line of memory. The cache controller 1878 in response to the asserted hit signal (hit0,...,hit7) generates a read enable

signal on the port (cache_read) for enabling reading of those cache-lines for which the hit signal has been asserted. When a request (proc_req) 1876 is asserted, but not the hit signal (hit0,...,hit7), a generated request (ext_req) 1890 is sent to the external memory together with the complete external
5 memory address for that cache-line of data. This cache-line is written into the eight banks of cache memory 230 via the input (ext_data) when it is available from the external memory. When this happens, the tag information is also written into the tag memory 1886 at that line address, and the line status bit 1873 for that line asserted.

10 Data from the eight banks of cache memory 230 is then outputted through a series of multiplexers in a data organizer 1892, so that data is positioned in a predetermined manner in an output data packet 1894. In one operational mode, the data organizer 1892 is able to select and output eight 8-bit words from the respective eight 32-bit words outputted from the eight
15 memory banks by utilising the current operational mode signal (dcc_mode) and the byte addresses (byte_addr) of the current generated complete external memory addresses. In another operational mode, the data organizer 1892 directly outputs the eight 32-bit words outputted from the eight memory banks. As noted previously, the data organizer arranges this data in a
20 predetermined manner for output.

A request would comprize the following steps:

- 1) The processing unit requests a packet of data by supplying an address to the processing unit interface of the cache controller 1878;
- 2) Each of the eight address generator units 1881 then generate a
25 separate address for each block of cache memory depending on the mode of operation;
- 3) The Tag portion of each of the generated addresses is then compared to the Tag address stored in the four blocks of triple-port Tag memory 1886 and addressed by each of the corresponding line part of the
30 eight generated addresses;

206

4) If they match, and the line valid status 1873 for that line is also asserted, the data requested for that block of memory is deemed to be resident in the said cache memory 230;

5) Data that is not resident is fetched via the external bus 1890 and all eight blocks of the cache memory 230 are updated with that line of data from external memory. The Tag address of the new data is then written to the Tag memory 1886 at the said line address, and the line valid status 1873 for that line asserted;

6) When all requested data items are resident in cache memory 230, it is presented to the processing unit in a predetermined packet format.

As previously noted, all the modules (Fig 2) of the coprocessor 224 include a standard cBus interface 303 (Fig 20). For more details on the standard cBus interface registers for the data cache controller 240 and cache 230, reference is made to pages B42 to B46 of Appendix B. The settings in these registers control the operation of the data controller 240. For the sake of simplicity only two of these registers are shown in Figure 153, i.e. base_address and dcc_mode.

Once the data cache controller 240 and data cache 230 are enabled, the data cache controller initially operates in the normal mode with all cache lines invalid. At the end of an instruction, the data cache controller 240 and cache 230 always reverts to the normal mode of operation. In all of the following modes except the "Invalidate" mode, there is an "Auto-fill and validate" option. By setting a bit in the dcc_cfg2 register, it is possible to fill the entire cache starting at the address stored in the base_address register. During this operation, the data requests from the operand organizers B and C 247,248 are locked out until the operation is complete. The cache is validated at the end of this operation.

a. Normal Cache Mode

In this mode, the two operand organizers supply the complete external memory addresses of the data requested. The address generator 1881

outputs the complete external memory addresses which are then checked independently using the internal tag memory 1872 to see that if the data requested is resident in the memory cache 230. If both requested data items are not in cache 230, data will be requested from the input interface switch

5 252. Round Robin scheduling will be implemented to service persistent simultaneous requests.

For simultaneous requests, if one of the data items is resident in cache, it will be placed on the least significant 32 bits of each requestor's data bus. The other data will be requested externally via the input interface switch.

10 **b. The Single Output General Color Space Conversion Mode**

In this mode, the request comes from operand organizer B in the form of a 12-bit byte address. The requested data items are 8-bit color output values as previously discussed with reference to Fig. 60. The 12-bit address is fed to the index_packet inputs of the address generators 1881 and the eight

15 address generators 1881 generate eight different 32-bit complete external memory addresses of the format shown in figure 96. The bank, line and byte addresses of the generated complete addresses are determined in accordance with Table 12 and Fig. 61. The external memory address is interpreted as eight 9-bit line and byte addresses, which are used to address a byte from

20 each of the eight banks of RAM. The cache is accessed to obtain the eight byte values from each bank which are returned to the operand organizers for subsequent interpolation by the main data path 242 in accordance with the principles previously discussed with reference to Fig. 60. As the single output color value table is able to fit entirely within the cache memory 230, it

25 is preferable to load the entire single output color value table within the cache memory 230 prior to enabling the single color conversion mode.

c. Multiple Output General Color Space Conversion Mode

In this mode, a 12-bit word address is received from operand organizer B

247. The requested data items are 32-bit color output values as previously

30 discussed with reference to Fig. 62. The 12-bit address is fed to the

index_packet inputs of the address generators 1881 and the eight address generators 1881 generate eight different 32-bit complete external memory addresses of the format shown in figure 96. The line and tag addresses of the complete external memory addresses are determined in accordance with table 12 and Fig. 63. The completed external memory address is interpreted as eight 9-bit addresses with the 9-bit address being decomposed into a 7-bit line address and a 2-bit tag address as discussed previously with reference to Fig. 63. Upon the tag address not being found, the cache stalls while the appropriate data is loaded from the input interface switch 252 (Fig 2). Upon the data being available, the output data is returned to the operand organizers.

d. JPEG Encoding Mode

In this mode, the necessary tables for JPEG encoding and other operational sub-sets are stored in each bank of cache RAM. The storage of tables being previously described in the previous discussion of the JPEG encoding mode (Tables 14 and 16).

e. Slow JPEG Decoding Mode

In this mode, the data is organized in accordance with Table 17.

f. Matrix Multiplication Mode

In this mode, the cache is utilized to access 256 byte lines of data.

g. Disabled Mode

In this mode, all requests are passed through to the input interface switch 252.

h. Invalidate Mode

In this mode, the contents of the entire cache are invalidated by clearing all the line valid status bits.

3.18.7 Input Interface Switch

Returning again to Fig. 2, the input interface switch 252 performs the function of arbitrating data requests from the pixel organizer 246, the data cache controller 240 and the instruction controller 235. Further, the input

interface switch 252 transmits addresses and data as required to the external interface controller 238 and local memory controller 236.

The input interface switch 252 stores in one of its configuration register the base address or the memory object in the host memory map. This is a virtual address that must be aligned on a page boundary, hence 20 address bits are required. For each request made by the pixel organizer, data cache controller, instruction controller, the input interface switch 252 first subtracts the co-processor's base address bits from the most significant 6 bits of the start address of the data. If the result is negative, or the most significant 6 bits of the result are non-zero, this indicates that the desired destination is the PCI bus.

If the most significant 6 bits of the result are zero, this indicates that the data maps to a co-processor's memory location. The input interface switch 252 then needs to check the next 3 bits to determine if the co-processor's location is legal or not.

The legal co-processor's locations that may act as a source of data are:

- 1) 16 Mbytes occupied by the Generic interface, beginning at an offset of 0x01000000 from the co-processor's base address.
- 2) 32 Mbytes occupied by the local memory controller (LMC), starting at an offset of 0x02000000 from the base address of the co-processor's memory object.

Requests that map to an illegal co-processor's location are flagged as errors by the Input Interface Switch.

The PCI bus is the source of data corresponding to any addresses that map outside of the range occupied by the co-processor's memory object. An i-source signal is used by the input interface switch to indicate to the EIC whether requested data is to originate from the PCI bus or the Generic interface.

After the address decoding process, legal requests are routed to the appropriate IBus interface when the bus is free. The EIC or LMC is busy with a data transaction to the input interface switch when they have their i-ack signal asserted. However, the input interface switch does not keep a count for the number of incoming words, and so must monitor the i-oe signal, controlled by the pixel organizer, instruction controller or data cache controller, in order to determine when the current data transaction has completed.

The input interface switch 252 must arbitrate between three modules: the pixel organizer, data cache controller and instruction controller. All of these modules are able to request data simultaneously, but not all requests can be instantly met since there are only two physical resources. The arbitration scheme used by the input interface switch is priority-based and programmable. Control bits within a configuration register of the input interface switch specify the relative priorities of the instruction controller, data cache controller and pixel organizer. A request from the module with the lower priority is granted when neither of the other two modules are requesting access to the same resource as it is. Assigning the same priority to at least two of the requesters results in the use of a round robin scheme to deduce the new winners.

As immediate access to a resource may not be possible, the input interface switch needs to store the address, burst length and whether to prefetch data provided by each requester. For any given resource, the arbitration process only needs to determine a new winner when there is not an IBus transaction in progress.

Turning to Fig. 145, there is illustrated the instruction interface switch 252 in more detail. The switch 252 includes the standard CBus interface and register file 860 in addition to two IBus transceivers 861 and 862 between an address decoder 863 and arbiter 864.

The address decoder 863 performs address decoding operations for requests received from the pixel organizer, data cache controller and

instruction controller. The address decoder 863 checks the address is a legal one and performs any address re-mapping required. The arbiter 864 decides which request to pass from one IBus transceiver 661 to a second IBus transceiver 862. Preferably, the priority system is programmable.

5 The IBus transceivers 861, 862 contain all the necessary multiplexing/demultiplexing and tristate buffering to enable communication over the various interfaces to the input interface switch.

3.18.8 Local Memory Controller

Returning again to Fig. 2, the local memory controller 236 is responsible for all aspects of controlling the local memory and handling access requests between the local memory and modules within the co-processor. The local memory controller 236 responds to write requests from the result organizer 249 and read requests from the input interface switch 252. Additionally, it also responds to both read and write requests from the peripheral interface controller 237 and the usual global CBus input. The local memory controller utilizes a programmable priority system and further utilizes FIFO buffers to maximize throughput.

In the present invention, a multi-port burst dynamic memory controller is utilized in addition to using First-In-First-Out (FIFO) buffers to de-couple the ports from a memory array.

Fig. 146 depicts a block diagram of a four-port burst dynamic memory controller according to a first embodiment of the present invention. The circuit includes two write ports (A 1944 and B 1946) and two read ports (C 1948 and D 1950) that require access to a memory array 1910. The data paths from the two write ports pass through separate FIFOs 1920, 1922 and to the memory array 1910 via a multiplexer 1912, while the data paths of the read ports 1948, 1950 pass from the memory array 1910 via separate FIFOs 1936, 1938. A central controller 1932 coordinates all port accesses as well as driving all the control signals necessary to interface to the dynamic memory 1910. A refresh counter 1934 determines when dynamic memory

refresh cycles for the memory array 1910 are required and coordinates these with the controller 1932.

Preferably, the data is read from and written to the memory array 1910 at twice the rate that data is transferred from the write ports 1944, 1946 to the FIFOs 1920, 1922 or from the FIFOs 1936, 1938 to the read ports 1948, 1950. This results in as little time as possible being taken up doing transfers to or from the memory array 1910 (which is the bottleneck of any memory system) relative to the time taken to transfer data through the write and read ports 1944, 1946, 1948, 1950.

Data is written into the memory array 1910 via either one of the write ports 1944, 1946. The circuits connected to the write ports 1944, 1946 see only a FIFO 1920, 1922 which are initially empty. Data transfers through the write ports 1944, 1946 proceed unimpeded until the FIFO 1920, 1922 is filled, or the burst is ended. When data is first written into the FIFO 1920, 1922, the controller 1932 arbitrates with the other ports for the DRAM access. When access is granted, data is read out of the FIFO 1920, 1922 at the higher rate and written into the memory array 1910. A burst write cycle to DRAM 1910 is only initiated when a preset number of data words have been stored in the FIFO 1920, 1922, or when the burst from the write port ends. In either case, the burst to DRAM 1910 proceeds when granted and continues until the FIFO 1920, 1922 is emptied, or there is a cycle request from a higher priority port. In either event, data continues to be written into the FIFO 1920, 1922 from the write port without hindrance, until the FIFO is filled, or until the burst ends and a new burst is started. In the latter case, the new burst cannot proceed until the previous burst has been emptied from the FIFO 1920, 1922 and written to the DRAM 1910. In the former case, data transfers recommences as soon as the first word is read out of the FIFO 1920, 1922 and written to DRAM 1910. Due to the higher rate of data transfers out of the FIFO 1920, 1922, it is only possible for the write port 1944, 1946 to stall if the controller 1932 is interrupted with cycle requests from the other ports.

Any interruption to the data transfers from the write ports 1944, 1946 to the FIFOs 1920, 1922 is preferably kept to a minimum.

The read ports 1948, 1950 operate in a converse fashion. When a read port 1948, 1950 initiates a read request, a DRAM cycle is immediately requested. When granted, the memory array 1910 is read and data is written into the corresponding FIFO 1936, 1938. As soon as the first data word is written into the FIFO 1936, 1938, it is available for read-out by the read port 1948, 1950. Thus there is an initial delay in obtaining the first datum word but after that there is a high likelihood that there are no further delays in retrieving the successive data words. DRAM reads will be terminated when a higher priority DRAM request is received, or if the read FIFO 1936, 1938 becomes full, or when the read port 1948, 1950 requires no more data. Once the read has been terminated in this way, it is not restarted until there is room in the FIFO 1936, 1938 for a preset number of data words. Once the read port terminates the cycle, any data remaining in the FIFO 1936, 1938 is discarded.

In order to keep DRAM control overheads to a minimum, re arbitration for the DRAM access is restricted so that bursts cannot be interrupted until a preset number of data words have been transferred (or until the corresponding write FIFO 1920, 1922 is emptied, or read FIFO 1936, 1938 is filled).

Each of the access ports 1944, 1946, 1948, 1950 has an associated burst start address which is latched in a counter 1942 at the start of the burst. This counter holds the current address for transactions on that port so that, should the transfer be interrupted, it can be resumed at any time at the correct memory address. Only the address for the currently active DRAM cycle is selected by multiplexer 1940 and passed on to the row address counter 1916 and column address counter 1918. The low order N bits of address are inputted to the column counter 1918 while the higher order address bits are inputted to the row counter 1916. Multiplexer 1914 outputs row addresses from the row counter 1916 to the memory array 1910 during the row address

time of the DRAM and passes column addresses from the column counter 1918 during column address time of the DRAM. The row address counter 1916 and the column address counter 1918 are loaded at the start of any burst to the memory array DRAM 1910. This is true both at the start of a port cycle and at the continuation of an interrupted burst. The column address counter 1918 is incremented after each transfer to memory has taken place while the row address counter 1916 is incremented when the column address counter 1918 rolls over to a count of zero. When the latter happens, the burst must be terminated and restarted at the new row address.

In the preferred embodiment it is assumed that memory array 1910 comprises 4 x 8 bit byte lines making up a 32 bits per word. Further there is associated with each write port 1944, 1946 a set of four byte write enable signals 1950, 1952 which individually allow data to be written to each 8-bit portion of each 32-bit data word in the memory array 1910. Since it is possible to arbitrarily mask the writing of data to any byte within each word that is written to the memory array 1910, it is necessary to store the write enable information along with each data word in corresponding FIFOs 1926, 1928. These FIFOs 1926, 1928 are controlled by the same signals that control the write FIFOs 1920, 1922 but are only 4 bits wide instead of the 32 bits required for the write data in FIFOs 1920, 1922. In like fashion, multiplexer 1930 is controlled in the same manner as the multiplexer 1912. The selected byte write enables are inputted to the controller 1932 which uses the information to selectively enable or disable writing to the addressed word in the memory array 1910 in synchronization with the write data being inputted to the memory array 1910 by way of multiplexer 1912.

The arrangement of Fig. 146 operates under the control of the controller 1932. Fig. 147 is a state machine diagram depicting the detail of operation of the controller 1932 of Fig. 146. After power up and at the completion of reset the state machine is forced into state IDLE 100 in which all DRAM control signals are driven inactive (high) and multiplexer 1914

drives row addresses to the DRAM array 1910. When a refresh or cycle request is detected, the transition is made to state RASDEL1 1962. On the next clock edge the transition to state RASDEL2 1964 is made. On the next clock edge, if the cycle request and refresh have gone away, the state machine returns to state IDLE 1900, otherwise, when the DRAM tRP (RAS precharge timing constraint) period has been satisfied, the transition to state RASON 1966 is made at which time the row address strobe signal, RAS, is asserted low. After tRCD (RAS to CAS delay timing constraint) has been satisfied, the transition to state COL 1968 is made, in which the multiplexer 1914 is switched over to select column addresses for inputting to the DRAM array 1910. On the next clock edge the transition to state CASON 1970 is made and the DRAM column address strobe (CAS) signal is driven active low. Once the tCAS (CAS active timing constraint) has been satisfied, the transition to state CASOFF 1972 is made in which the DRAM column address strobe (CAS) is driven inactive high once again. At this point, if further data words are to be transferred and a higher priority cycle request or refresh is not pending or if it is too soon to re-arbitrate anyway, and once the tCP (CAS precharge timing constraint) has been satisfied, the transition back to state CASON 1970 will be made in which the DRAM column address strobe (CAS) is driven active low again. If no further data words are to be transferred, or if re-arbitrating is taking place and a higher priority cycle request or refresh is pending, then the transition is made to state RASOFF 1974 instead, providing tRAS (RAS active timing constraint) and tCP (CAS precharge timing constraint) are both satisfied. In this state the DRAM row address strobe (RAS) signal is driven inactive high. On the next clock edge the state machine returns to state IDLE 1860 ready to start the next cycle.

When in state RASDEL2 1964 and a refresh request is detected, the transition will be made to state RCASON 1980 once tRP (RAS precharge timing constraint) has been satisfied. In this state DRAM column address strobe is driven active low to start a DRAM CAS before RAS refresh cycle.

On the next clock edge the transition to state RRASON 1978 is made in which DRAM row address strobe (RAS) is driven active low. When tCAS (CAS active timing constraint) has been met, the transition to state RCASOFF 1976 will be made in which DRAM column address strobe (CAS) is driven inactive high. Once tRAS (RAS active timing constraint) has been met, the transition to state RASOFF 1974 is made in which DRAM row address strobe (RAS) is driven inactive high effectively ending the refresh cycle. The state machine then continues as above for a normal DRAM cycle, making the transition back to state IDLE 1960.

The refresh counter 1934 of Fig. 146 is simply a counter that produces refresh request signals at a fixed rate of once per 15 microseconds, or other rate as determined by the particular DRAM manufacturer's requirements. When a refresh request is asserted, it remains asserted until acknowledged by the state machine of Fig. 147. This acknowledgement is made when the state machine enters state RCASON 1980 and remains asserted until the state machine detects the refresh request has been de-asserted.

In Fig. 148, there is set out in pseudo code form, the operation of the arbitrator 1924 of Fig. 146. It illustrates the method of determining which of four cycle requesters is granted access to the memory array 1910, and also a mechanism for modifying the cycle requester priorities in order to maintain a fair access regime. The symbols used in this code are explained in Fig. 149.

Each requester has 4 bits associated with it that represent that requester's priority. The two high order bits are preset to an overall priority by way of configuration values set in a general configuration register. The two low order bits of priority are held in a 2-bit counter that is updated by the arbitrator 24. When determining the victor in an arbitration, the arbitrator 1924 simply compares the 4-bit values of each of the requesters and grants access to the requester with the highest value. When a requester is granted a cycle its low order 2-bit priority count value is cleared to zero, while all other requesters with identical high order 2-bit priority values and whose low order

2-bit priority is less than the victor's low order 2-bit priority have their low order 2-bit priority counts incremented by one. This has the effect of making a requester that has just been granted access to the memory array 1910 the lowest priority among requesters with the same priority high order 2-bit value. The priority low order 2-bit value of other requesters with priority high order 2-bit value different to that of the winning requester are not affected. The high order two bits of priority determine the overall priority of a requester while the low order two bits instill a fair arbitration scheme among requesters with identical high order priority. This scheme allows a number of arbitration schemes to be implemented ranging from hard-wired fixed priority (high order two bits of each requester unique) through part rotating and part hard-wired (some high order 2-bit priorities different to others, but not all) to strictly fair and rotating (all priority high order 2-bit fields the same).

Fig. 149 depicts the structure of the priority bits associated with each requester and how the bits are utilized. It also defines the symbols used in Fig. 148.

In the preferred embodiment, the various FIFOs 1920, 1922, 1938 and 1936 are 32 bits wide and 32 words deep. This particular depth provides a good compromise between efficiency and circuit area consumed. However, the depth may be altered, with a corresponding change in performance, to suit the needs of any particular application.

Also, the four port arrangement shown is merely a preferred embodiment. Even the provision of a single FIFO buffer between the memory array and either a read or write port will provide some benefits. However, the use of multiple read and write ports provides the greatest potential speed increase.

3.18.9 Miscellaneous Module

The miscellaneous module 239 provides clock generation and selection for the operation of the co-processor 224, reset synchronization, multiplexing of error and interrupt signals by routing of internal diagnostic signals to

external pins as required, interfacing between the internal and external form of the CBus and multiplexing of internal and generic Bus signals onto a generic/external CBus output pins. Of course, the operation of the miscellaneous module 239 varies in accordance with clocking requirements and implementation details depending on the ASIC technology utilized.

3.18.10 External Interface Controller

The following described aspects of the invention relate to a method and an apparatus for providing virtual memory in a host computer system having a co-processor that shares the virtual memory. The embodiments of the invention seek to provide a co-processor able to operate in a virtual memory mode in conjunction with the host processor.

In particular, the co-processor is able to operate in a virtual memory mode of the host processor. The co-processor includes a virtual-memory-to-physical-memory mapping device that is able to interrogate the host processor's virtual memory tables, so as to map instruction addresses produced by the co-processor into corresponding physical addresses in the host processor's memory. Preferably, the virtual-memory-to-physical-memory mapping device forms part of a computer graphics co-processor for the production of graphical images. The co-processor may include a large number of modules able to form various complex operations on images. The mapping device is responsible for the interaction between the co-processor and the host processor.

The external interface controller (EIC) 238 provides the co-processors interface to the PCI Bus and to a generic Bus. It also provides memory management to translate between the co-processor's internal virtual address space and the host system physical address space. The external interface controller 238 acts as a master on the PCI Bus when reading the data from the host memory in response to a request from the input interface switch 252 and when writing data to host memory in response to a request from the result organizer 249. The PCI Bus access is implemented in accordance the

well known standard with "PCI Local Bus specification, draft 2.1", PCI special interest group, 1994.

The external interface controller 238 arbitrates between simultaneous requests for PCI transactions from the input interface switch 252 and the result organizer 249. The arbitration is preferably configurable. The types of requests received include transactions for reading less than one cache line of the host co-processor at a time, reading between one and two cache lines of the host and reading two or more cache lines of the host. Unlimited length write transactions are also implemented by the external interface controller 238. Further, the external interface controller 238 optionally also performs prefetching of data.

The construction of the external interface controller 238 includes a memory management unit which provides virtual to physical address mapping of host memory accesses for all of the co-processor's internal modules. This mapping is completely transparent to the module requesting the access. When the external interface controller 238 receives a request for host memory access, it initiates a memory management unit operation to translate the requested address. Where the memory management unit is unable to translate the address, in some cases this results in one or more PCI Bus transaction to complete the address translation. This means that the memory management unit itself can be another source of transaction requests on the PCI Bus. If a requested burst from the input interface switch 252 or results organizer 249 crosses the boundary of a virtual page, the external interface controller 238 automatically generates a memory management unit operation to correctly map all virtual addresses.

The memory management unit (MMU) (915 of Fig. 150) is based around a 16 entry translation look aside buffer (TLB). The TLB acts as a cache of virtual to physical address mappings. The following operations are possible on the TLB:

1) Compare: A virtual address is presented, and the TLB returns either the corresponding physical address, or a TLB miss signal (if no valid entry matches the address).

2) Replace: A new virtual-to-physical mapping is written into the TLB, replacing an existing entry or an invalid entry.

3) Invalidate: A virtual address is presented; if it matches a TLB entry, that entry is marked invalid.

4) Invalidate All. All TLB entries are marked invalid.

5) Read: A TLB entry's virtual or physical address is read, based on a four bit address. Used for testing only.

6) Write: A TLB entry's virtual and physical address is written, based on a four bit address.

Entries within the TLB have the format shown in Fig. 151. Each valid entry consists of a 20-bit virtual address 670, a 20-bit physical address 671, and a flag which indicates whether the corresponding physical page is writable. The entries allow for page sizes as small as 4kB. A register in the MMU can be used to mask off up to 10 bits of the addresses used in the comparison. This allows the TLB to support pages up to 4MB. As there is only one mask register, all TLB entries refer to pages of the same size.

The TLB uses a "least-recently-used" (LRU) replacement algorithm. A new entry is written over the entry which has the longest elapsed time since it was last written or matched in a comparison operation. This applies only if there are no invalid entries; if these exist, they are written to before any valid entries are overwritten.

Fig. 152 shows the flow of a successful TLB compare operation. The incoming virtual address 880 is divided into 3 parts 881 - 883. The lower 12 bits 881 are always part of the offset inside a page and so are passed directly on to the corresponding physical address bits 885. The next 10 bits 882 are either part of the offset, or part of the page number, depending on the page size, as set by the mask bits. A zero in the mask register 887 indicates that

the bit is part of the page offset, and should not be used for TLB comparisons. The 10 address bits are logically "ANDED" with the 10 mask bits to give the lower 10 bits of the virtual page number 889 for TLB lookups. The upper 10 bits 883 of the virtual address are used directly as the upper 10 bits of the virtual page number 889.

The 20-bit virtual page number thus generated is driven into the TLB. If it matches one of the entries, the TLB returns the corresponding physical page number 872, and the number of the matched location. The physical address 873 is generated from the physical page number using the mask register 887 again. The top 10 bits of physical page number 872 are used directly as the top 10 bits of the physical address 873. The next 10 bits of physical address 872 are chosen 875 from either the physical page number (if the corresponding mask bit is 1), or the virtual address (if the mask bit is 0). The lower 12 bits 885 of physical address come directly from the virtual address.

Finally, following a match, the LRU buffer 876 is updated to reflect the use of the matched address.

A TLB miss occurs when the input interface switch 252 or the results organizer 249 requests an access to a virtual address which is not in the TLB 872. In this case, the MMU must fetch the required virtual-to-physical translation from the page table in host memory 203 and write it into the TLB before proceeding with the requested access.

The page table is a hash table in the hosts main memory. Each page table entry consists of two 32-bit words, with the format shown in Fig. 153. The second word comprizes the upper 20 bits for the physical address and the lower 12 bits are reserved. The upper 20 bits of the corresponding virtual address are provided in the first word. The lower 12 bits include a valid (V) bit and writable (W) or a "read-only" bit, with the remaining 10 bits being reserved.

The page table entry contains essentially the same information as the TLB entry. Further flags in the page table are reserved. The page table itself may be, and typically is, distributed over multiple pages in main memory 203, which in general are contiguous in virtual space but not physical space.

The MMU contains a set of 16 page table pointers, setup by software, each of which is a 20-bit pointer to a 4kB memory region containing part of the page table. This means the co-processor 224 supports a page table 64kB in size, which holds 8k page mappings. For systems with a 4kB page size, this means a maximum of 32MB of mapped virtual address space. Preferably, the page table pointers always reference a 4kB memory region, regardless of the page size used in the TLB.

The operation of the MMU following a TLB miss is shown 690 in Fig. 154, as follows:

1. Execute the hash function 892 on the virtual page number 891 that missed in the TLB, to produce a 13-bit index into the page table.
2. Use the top 4 bits 894 of the page table index 894, 896 to select a page table pointer 895.
3. Generate the physical address 890 of the required page table entry, by concatenating the 20-bit page table pointer 895 with the lower 9 bits of the page table index 896, setting the bottom 3 bits to 000 (since page table entries occupy 8 bytes in host memory).
4. Read 8 bytes from host memory, starting at the page table entry physical address 898.
5. When the 8-byte page table entry 900 is returned over the PCI bus, the virtual page number is compared to the original virtual page number that caused the TLB miss, provided that the VALID bit is set to 1. If it does not match, the next page table entry is fetched (incrementing the physical address by 8 bytes) using the process described above. This continues until a page table entry with a matching virtual page number is found, or an invalid

page table entry is found. If an invalid page table entry is found, a page fault error is signalled and processing stops.

6. When a page table entry with a matching virtual page number is found, the complete entry is written into the TLB using the replace operation. The new entry is placed in the TLB location pointed to by the LRU buffer 876.

The TLB compare operation is then retried, and will succeed, and the originally requested host memory access can proceed. The LRU buffer 876 is updated when the new entry is written into the TLB.

10 The hash function 892 implemented in the EIC 238 uses the following equation on the 20 bits of virtual page number (vpn):

$$\text{index} = ((\text{vpn} \gg S_1) \text{ XOR } (\text{vpn} \gg S_2) \text{ XOR } (\text{vpn} \gg S_3)) \& 0\text{x}1\text{fff};$$

where s_1 , s_2 and s_3 are independently programmable shift amounts (positive or negative), each of which can take on four values.

15 If the linear search through the page table crosses a 4kB boundary, the MMU automatically selects the next page table pointer to continue the search at the correct physical memory location. This includes wrapping around from the end of the page table to the start. The page table always contains at least one invalid (null) entry, so that the search always terminates.

20 Whenever the software replaces a page in host memory, it must add a page table entry for the new virtual page, and remove the entry corresponding to the page that has been replaced. It must also make sure that the old page table entry is not cached in the TLB on the co-processor 224. This is achieved by performing a TLB invalidation cycle in the MMU.

25 An invalidation cycle is performed via a register write to the MMU, specifying the virtual page number to be invalidated, along with a bit that causes the invalidation operation to be done. This register write may be performed directly by the software, or via an instruction interpreted by the Instruction Decoder. An invalidation operation is performed on the TLB for
30 the supplied virtual page number. If it matches a TLB entry, that entry is

marked invalid, and the LRU table updated so that the invalidated location is used for the next replace operation.

A pending invalidate operation has priority over any pending TLB compares. When the invalidate operation has completed, the MMU clears
 5 the invalidate bit, to signal that it can process another invalidation.

If the MMU fails to find a valid page table entry for a requested virtual address, this is termed a page fault. The MMU signals an error, and stores the virtual address that caused the fault in a software accessible register. The MMU goes to an idle state and waits until this error is cleared.
 10 When the interrupt is cleared, the MMU resumes from the next requested transaction.

A page fault is also signalled if a write operation is attempted to a page that is (not marked writable) marked read only.

The external interface controller (EIC) 238 can service transaction
 15 requests from the input interface switch 252 and the result organizer 249 that are addressed to the Generic bus. Each of the requesting modules indicates whether the current request is for the Generic Bus or the PCI bus. Apart from using common buses to communicate with the input interface switch 252 and the results organizer 249, the EIC's operation for Generic bus requests is
 20 entirely separate from its operation for PCI requests. The EIC 238 can also service CBus transaction types that address the Generic bus space directly.

Fig. 150 shows the structure of the external interface controller 238. The IBus requests pass through a multiplexer 910, which directs the requests to the appropriate internal module, based on the destination of the request
 25 (PCI or Generic Bus). Requests to the Generic bus pass on to the generic bus controller 911, which also has RBus and CBus interfaces. Generic bus and PCI bus requests on the RBus use different control signals, so no multiplexer is required on this bus.

IBus requests directed to the PCI bus are handled by an IBus Driver
 30 (IBD) 912. Similarly, an RBus Receiver (RBR) 914 handles the RBus

requests to PCI. Each of the IBD 912 and RBR 914 drive virtual addresses to the memory management unit (MMU) 915, which provides physical addresses in return. The IBD, RBR and MMU can each request PCI transactions, which are generated and controlled by the PCI master mode controller (PMC) 917. The IBD and the MMU request only PCI read transactions, while the RBR requests only PCI write transactions.

A separate PCI Target Mode Controller (PTC) 918 handles all PCI transactions addressed to the co-processor as a target. This drives CBus master mode signals to the instruction controller, allowing it to access all other modules. The PTC passes returned CBus data to be driven to the PCI bus via the PMC, so that control of the PCI data bus pins comes from a single source.

CBus transactions addressed to EIC registers and module memory are dealt with by a standard CBus interface 7. All submodules receive some bits from control registers, and return some bits to status registers, which are located inside the standard CBus interface.

Parity generation and checking for PCI bus transactions is handled by the parity generate and check (PGC) module 921, which operates under the control of the PMC and PTC. Generated parity is driven onto the PCI bus, as are parity error signals. The results of parity checking are also sent to the configuration registers section of the PTC for error reporting.

Fig. 155 illustrates the structure of the IBus driver 912 of Fig. 150. Incoming IBus address and control signals are latched 930 at the start of a cycle. An or-gate 931 detects the start of the cycle and generates a start signal to control logic 932. The top address bits of the latch 930, which form the virtual page number, are loaded into a counter 935. The virtual page number is passed to the MMU 915 (Fig. 150) which returns a physical page number which is latched 936.

The physical page number and the lower virtual address bits are recombined according to the mask 937 and form the address 938 for PCI

requests to the PMC 717 (Fig.102). The burst count for the cycle is also loaded into a counter 939. Prefetch operations use another counter 941 and an address latch and compare circuit 943.

5 Data returned from the PMC is loaded into a FIFO 944, along with a marker which indicates whether the data is part of a prefetch. As data becomes available at the front of the FIFO 944, it is clocked out by the read logic via synchronization latches 945,946. The read logic 946 also generates the IBus acknowledge signal.

10 A central control block 932, including state machines, controls the sequencing of all of the address and data elements, and the interface to the PMC.

The virtual page number counter 935 is loaded at the start of an IBus transaction with the page number bits from the IBus address. The top 10 bit of this 20-bit counter always come from the incoming address. For the lower 15 10 bits, each bit is loaded from the incoming address if the corresponding mask bit 937 is set to 1; otherwise, the counter bit is set to 1. The 20-bit value is forwarded to the MMU interface.

In normal operation the virtual page number is not used after the initial address translation. However, if the IBD detects that the burst has 20 crossed a page boundary, the virtual page counter is incremented, and another translation is performed. Since the low order bits that are not part of the virtual page number are set to 1 when the counter is loaded, a simple increment on the entire 20-bit value always causes the actual page number field to increment. The mask bits 937 are used again after an increment to 25 set up the counter for any subsequent increments.

The physical address is latched 936 whenever the MMU returns a valid physical page number after translation. The mask bits are used to correctly combine the returned physical page number with the original virtual address bits.

The physical address counter 938 is loaded from the physical address latch 936. It is incremented each time a word is returned from the PMC. The count is monitored as it increments, to determine whether the transaction is about to cross a page boundary. The mask bits are used to determine which bits of the counter should be used for the comparison. When the counter detects that there are two or less words remaining in the page, it signals the control logic 932, which terminates the current PCI request after two more data transfers, and requests a new address translation if required. The counter is reloaded after the new address translation, and PCI requests resumed.

The burst counter 939 is a 6-bit down counter which is loaded with the IBus burst value at the beginning of a transaction. It is decremented every time a word is returned from the PMC. When the counter value is two or less, it signals to the control logic 932, which can then terminate the PCI transaction correctly with two more data transfers (unless prefetching is enabled).

The prefetch address register 943 is loaded with the physical address of the first word of any prefetch. When the subsequent IBus transaction starts, and the prefetch counter indicates that at least one word was successfully prefetched, the first physical address of the transaction is compared to the value in the prefetch address latch. If it matched, the prefetch data is used to satisfy the IBus transaction, and any PCI transaction requests start at the address after the last prefetched word.

The prefetch counter 941 is a four bit counter which is incremented whenever a word is returned by the PMC during a prefetch operation, up to a maximum count equal to the depth of the input FIFO. When the subsequent IBus transaction matches the prefetch address, the prefetch count is added to the address counter, and subtracted from the burst counter, so that PCI requests can start at the required location. Alternatively, if the IBus transaction only requires some of the prefetched data, the requested burst

length is subtracted from the prefetch count, and added to the latched prefetch address, and the remaining prefetch data is retained to satisfy further requests.

The Data FIFO 944 is a 8 word by 33 bit asynchronous fall through
5 FIFO. Data from the PMC is written into the FIFO, along with a bit indicating whether the data is part of a prefetch. Data from the front of the FIFO is read out and driven onto the IBus as soon as it becomes available. The logic that generates the data read signals operates synchronously to clk, and generates the IBus acknowledge output. If the transaction is to be
10 satisfied using prefetched data, signals from the control logic tell the read logic how many words of prefetched data should be read out of the FIFO.

Fig. 156 illustrates the structure of the RBus Receiver 914 of Fig. 150. Control is split between two state machines 950, 951. The Write state machine 951 controls the interface to the RBus. The input address 752 is
15 latched at the start of an RBus burst. Each data word of the burst is written in a FIFO 754, along with its byte enables. If the FIFO 954 become full r-ready is deasserted by the write logic 951 to prevent the results organiser from attempting to write any more words.

The write logic 951 notifies the main state machine 950 of the start of
20 an RBus burst via a resynchronized start signal to prevent the results organizer from trying to write any more words. The top address bits, which form the virtual page number, are loaded into a counter 957. The virtual page number is passed to the MMU, which returns a physical page number 958. The physical page number and the lower bits of the virtual address are
25 recombined according to the mask, and loaded into a counter 960, to provide the address for PCI requests to the PMC. Data and byte enables for each word of the PCI request are clocked out of the FIFO 954 by the main control logic 950, which also handles all PMCM interface control signals. The main state machine indicates that it is active via a busy signal, which is
30 resynchronized and returned to the write state machine.

The write state machine 951 detects the end of an RBus burst using r-final. It stops loading data into the FIFO 954, and signals the main state machine that the RBus burst has finished. The main state machine continues the PCI requests until the Data FIFO has been emptied. It then
5 deasserts busy, allowing the write state machine to start the next RBus burst.

Returning to Fig. 150, the memory management unit 915 is responsible for translating virtual page numbers into physical page numbers for the IBus driver (IBD) 912 and the RBus receiver (IBR) 914. Turning to
10 Fig. 157, there is illustrated the memory management unit in further detail. A 16 entry translation lookaside buffer (TLB) 970 takes its inputs from, and drives its outputs to, the TLB address logic 971. The TLB control logic 972, which contains a state machine, receives a request, buffered in the TLB address logic, from the RBR or IBD. It selects the source of the inputs, and
15 selects the operation to be performed by the TLB. Valid TLB operations are compare, invalidate, invalidate all, write and read. Sources of TLB input addresses are the IBD and RBR interfaces (for compare operations), the page table entry buffer 974 (for TLB miss services) or registers within the TLB address logic. The TLB returns the status of each operation to the TLB
20 control logic. Physical page numbers from successful compare operations are driven back to the IBD and RBR. The TLB maintains a record of its least recently used (LRU) location, which is available to the TLB address logic for use as a location for write operations.

When a compare operations fails, the TLB control logic 972 signals
25 the page table access control logic 976 to start a PCI request. The page table address generator 977 generates the PCI address based on the virtual page number, using its internal page table pointer registers. Data returned from the PCI request is latched in the page table entry buffer 974. When a page table entry that matches the required virtual address is found, the physical
30 page number is driven to the TLB address logic 977 and the page table access

control logic 976 signals that the page table access is complete. The TLB control logic 972 then writes the new entry into the TLB, and retrics the compare operation.

Register signals to and from the SCI are resynchronized 980 in both
5 directions. The signals go to and from all other submodules. A module memory interface 981 decodes access from the Standard CBus Interface to the TLB and page table pointer memory elements. TLB access are read only, and use the TLB control logic to obtain the data. The page table pointers are read/write, and are accessed directly by the module memory interface. These
10 paths also contain synchronization circuits.

3.18.11 Peripheral Interface Controller

Turning now to Fig. 158, there is illustrated one form of peripheral interface controller (PIC) 237 of Fig. 2 in more detail. The PIC 237 works in one of a number of modes to transfer data to or from an external peripheral
15 device. The basic modes are:

1) Video output mode. In this mode, data is transferred to a peripheral under the control of an external video clock and clock/data enables. The PIC 237 drives output clock and clock enable signs with the required timing with respect to the output data.

20 2) Video input mode. In this mode, data is transferred from a peripheral under the control of an external video clock and data enable.

3) Centronics mode. This mode transfers data to and from the peripheral according to the standard protocol defined in IEEE 1284 standard.

The PIC 237 decouples the protocol of the external interface from the
25 internal data sources or destination in accordance with requirements. Internal data sources write data into a single stream of output data, which is then transferred to the external peripheral according to the selected mode. Similarly, all data from an external peripheral is written into a single input data stream, which is available to satisfy a requested transaction to either of
30 the possible internal data destinations.

There are three possible sources of output data: the LMC 236 (which uses the ABus), the RO 249 (which uses the RBus), and the global CBus. The PIC 237 responds to transactions from these data sources one at a time - a complete transaction is completed from one source before another source is considered. In general, only one source of data should be active at any time. If more than one source is active, they are served with the following priority - CBus, then ABus, then RBus.

As usual, the module operates under the control of the standard CBus interface 990 which includes the PIC's internal registers.

Further, a CBus data interface 992 is provided for accessing and controlling peripheral devices via the co-processor 224. An ABus interface 991 is also provided for handling memory interactions with the local memory controller. Both the ABus interface 991 and CBus data interface 992 in addition to the result organizer 249 send data to an output data path 993 which includes a byte - wide FIFO. Access to the output data path is controlled by an arbiter which keeps track of which source has priority or ownership of the output stream. The output data path in turn interfaces with a video output controller 994 and centronics control 997 depending on which of these is enabled. Each of the modules 994, 997 reads one byte at a time from the output data path's internal FIFO. The centronics controller 997 implements the centronics data interfacing standard for controlling peripheral devices. The video output controller includes logic to control output pads according to the desired video output protocols. Similarly, a video input controller 998 includes logic to control any implemented video input standard. The video input controller 998 outputs to an input data path unit 999 which again comprizes a byte wide input FIFO with data being written into the FIFO asynchronously, one byte at a time, by either the video input controller 998 or centronics controller 997.

A data timer 996 contains various counters utilized to monitor the current state of FIFO's within output data paths 993 and input data path 999.

It can be seen from the foregoing that the co-processor can be utilized to execute dual streams of instructions for the creation of multiple images or multiple portions of a single image simultaneously. Hence, a primary instruction stream can be utilized to derive an output image for a current page while a secondary instruction stream can be utilized, during those times when the primary instruction stream is idle, to begin the rendering of a subsequent page. Hence, in a standard mode of operation, the image for a current page is rendered and then compressed utilising the JPEG coder 241. When it is required to print out the image, the co-processor 241 decompresses the JPEG encoded image, again utilising the JPEG coder 241. During those idle times when no further portions of the JPEG decoded image are required by an output device, instructions can be carried out for the compositing of a subsequent page or band. This process generally accelerates the rate at which images are produced due to the overlap operating of the co-processor. In particular, the co-processor 224 can be utilized to substantial benefit in the speeding up of image processing operations for printing out by a printer attached to the co-processor such that rendering speeds will be substantially increased.

It will be evident from the foregoing that discussion of the preferred embodiment refers to only one form of implementation of the invention and modifications, obvious to those skilled in the art, can be made thereto without departing from the scope of the invention.

Appendix A: The Coprocessor Microprogramming

This section details the actions performed internally in the coprocessor each time a new instruction is executed. All self-configuration carried out by the coprocessor during instruction execution is performed via internal register reads and writes, consequently the coprocessor can be fully microprogrammed via the External Cbus Interface or by the host via the PCIbus Interface. Note that microprogramming via the host interface will typically be very difficult due to the problems of synchronising the host.

In this Section it is assumed that the reader is familiar with the coprocessor's:

1. execution model,
2. instruction set and its coding,
3. register set,
4. and internal structure..

A.1 General Notes

A.1.1 General Coprocessor Setup Principles

For all instructions other than Control Instructions and Local DMA Instructions, the data flow through the coprocessor is fundamentally under the control of the Pixel Organiser. The Pixel Organiser is responsible for fetching the primary input stream of data, for counting this flow of data and for determining when the final piece of data has been fetched. Generally speaking, the other modules within the coprocessor simply respond to data sent to them.

A.1.2 Module Configuration Order

Not all modules are set up for every instruction. Some modules are never configured in the course of instruction decoding. The order that modules are configured is always the same and is PO, DCC, DOB, OOC, MDP, JC, RO, PIC.

A.1.3 Setting miscellaneous registers

If an instruction is encoded to include the setting of a register value, then that register is set during microprogramming in sequence defined by the following rules:

1. If the register to be set is in a module that would otherwise not have any registers set, then that register is set prior to any other register settings
2. If the register to be set is in a module that will have other registers set, then the extra register is set after other registers but immediately prior to that module's `_cfg` register.

A.1.4 Inconsistent Instruction Operand Coding

Many instructions have implied data types for operands and results, with other data types producing meaningless output. For each of the operands, the coprocessor determines the intended format using the following procedure:

1. if the internal format of the operand is specified to be one of pixels, packed bytes or unpacked bytes, then the corresponding Operand Organiser will be set up to reflect this. The Data Cache Controller will not be configured and will thus continue to operate in "normal" mode.
2. if the internal format of the operand is specified to be "other" then the coprocessor will derive the format of the operand from the instruction. For Operand B and Operand C, this is straight forward. For Operand A there is no naturally implied "other" format, and the coprocessor's behaviour is undefined. The corresponding Operand Organiser will be left

in bypass mode, and the Data Cache Controller will be set up to manage the operand data in the derived format.

Microprogramming is reasonably orthogonal split between the various modules.

A.1.5 Pseudocode syntax

- The sequence of operations is given by the numbers in the left margin.
- Register names are in *helvetica bold*.
- Register fields are denoted *register.field*
- I, D = Instruction word and Data word respectively of instruction currently being decoded
- A, B and C = A Operand word, B Operand word and C Operand word of instruction currently being decoded
- A_descriptor, B_descriptor and C_descriptor = A descriptor, B descriptor and C descriptor of Data Word of instruction currently being decoded
- R = Result word of instruction currently being decoded
- "X:Y" = X concatenated with Y
- "ex" = the coprocessor register number X
- "Cbus(X)" = do CBus operation X
- "*Cbus(X)" = data received from CBus read operation X
- "*X" = virtual memory location X
- "??" = unknown value, yet to be determined
- "set" refers to the setting of the Data Manipulation Registers.

A.2 Compositing Operators

Notes

1. Major opcode = 0xC and 0xD
2. Opacity is considered to be the byte of highest address i.e. is the most significant byte.
3. Either the accumulator or the operand may be pre-multiplied
4. Result may be un-pre-multiplied
5. Instruction length is defined in terms of the number of input pixels

```
-- Pixel Organiser:
if I.R=0 then
1.   po_len <- 0x0000:I.length
endif
if A_descriptor.S=0 then
2.   po_dmr <- set
endif
3.   po_said <- R
4.   po_cfg.mode <- sequential          -- set going
   po_cfg.dst <- mdp

-- DCC in "normal32" mode (default)

-- Operand Organiser B:
if B_descriptor.If != other then      -- not blend
5.   oob_len <- po_len
   if B_descriptor.S = 0 then
6.   oob_dmr <- set
   endif
7.   oob_said <- A                      -- special for compositing
   if B_descriptor.what = tile
8.   oob_tlla <- B
   endif
```

```

9.      oob_cfg.operate <- operate
      endif

      -- Operand Organiser C
      -- There is no implied other data type for this
      -- bo used to specify any bit offset in a bit map_attenuation
      if C_descriptor.if != other then
10.      ooc_len <- po_len
          -- set up the ooc_dmr
          if C_descriptor.S = 0 then
11.      ooc_dmr <- set
          endif
          -- including bo
12.      ooc_said <- C
13.      ooc_cfg.operate <- operate
      endif

      -- Main Data Path:
      if B_descriptor.if = other then
14.      mdp_len <- po_len
15.      mdp_bm <- B
16.      mdp_bl <- A
      end if
17.      mdp_cfg.instruction <- I.opcode
      mdp_cfg.blendgen <- (B_descriptor.if = other)
      mdp_cfg.decode <- 1
      mdp_cfg.operate <- 1

      -- Result Organiser:
      if R_descriptor.S = 0 then
18.      ro_dmr <- set
      endif
19.      ro_sa <- R
20.      ro_cfg.mode <- sequential

```

A.3 Colour Space Conversion

Notes

1. Input space is always three dimensions. By default this is the three least significant channels of the pixel - the opacity is the one thrown away.
2. The colour table format is one of two types: containing either one output channel or four output channels.

```

-- Pixel Organiser:
-- Operand A only makes sense as source pixels, anything else
-- is probably wrong
if I.R = 0 then
1.      po_len <- 0x0000:1.len
      endif
      if A_descriptor.S = 0 then
2.      po_dmr <- set
      endif
3.      po_said <- A
4.      po_muv <- C
5.      po_cfg.mode <- CSC
      po_cfg.dst <- mdp
          -- I & F tables

```

```

-- DCC setup:
-- should be other
if D_descriptor.if = other then
6.   dcc_addr <- B
7.   dcc_cfg2.cache_miss_inst <- B_descriptor.C
   if LM = 0 then -- single output channel
       dcc_cfg2.mode <- single channel CSC
   else
       dcc_cfg2.mode <- multi channel CSC
   endif
endif

-- Operand Organiser B:
-- Operand B should be other, anything else is probably
-- wrong but do it anyway
if B_descriptor.If != other then
8.   oob_len <- po_len
   if B_descriptor.S = 0 then
9.       oob_dmr <- set
   endif
10.  oob_said <- B
11.  oob_cfg.operate <- operate
endif

-- Operand Organiser C:
-- Operand C should be other, anything else is probably
-- wrong but do it anyway
if C_descriptor.If != other then
12.  ooc_len <- po_len
   if C_descriptor.S = 0 then
13.       ooc_dmr <- set
   endif
14.  ooc_said <- C
15.  ooc_cfg.operate <- operate
endif

-- Main Data Path:
16.  mdp_cfg.instruction <- I.opcode
   mdp_cfg.decode <- 1
   mdp_cfg.operate <- 1

-- Result Organiser:
if R_descriptor.S = 0 then
17.  ro_dmr <- set
endif
18.  ro_sa <- R
19.  ro_cfg.mode <- sequential

```

A.4 JPEG Instructions

Notes

1. Opcode = 0x2
2. Operand C may be a register to set
3. Options are many:

- subsampling or not
- filtering or not
- 1, 3, or 4 scan
- 4. These instructions rely on several registers to be set up prior to the instruction being executed.

A.4.1 Decompression

Notes

1. The following registers should be set prior to this instruction being executed:
 - `ro_idr`: output image dimensions register
 - `ro_out`: output cur register
 - `ro_lmt`: output limit register
- if there is a register set targeted at anything other than
-- the JC, PO, RO or DCC
if (D.R = 1) and (C_descriptor.topnibble != 0x5, 0x6, 0x9 or 0xB) then
 0. C_descriptor <- C
 endif
 - Pixel Organiser:
-- Operand A is the source compressed data, and should be a
-- byte stream
if I.R = 0 then
 1. po_len <- 0x0000:llen
 endif
 - if A_descriptor.S = 0 then
 2. po_dmr <- set
 endif
 3. po_said <- A
 -- if there is a register set targeted at PO
 if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
 4. C_descriptor <- C
 endif
 5. po_cfg.mode <- sequential
 po_cfg.dst <- jc
 - DCC setup:
if B_descriptor.If = other then
 6. dcc_addr <- B
 -- if there is a register set targeted at DCC
 if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
 7. C_descriptor <- C
 endif
 8. dcc_cfg2.C <- B_descriptor.C
 dcc_cfg2.mode <- JPEG decode
 end if
 - Operand Organiser B:
-- Operand B should be other, anything else is probably
-- wrong, but do it anyway
if B_descriptor.If != other then
 9. oob_len <- po_len
 if B_descriptor.S = 0 then
 10. oob_dmr <- set
 endif

```

11.    oob_said <- B
12.    oob_cfg.operate <- operate
    endif

    -- Operand Organiser C:
    -- Operand C may be a register to set. If it isn't it should
    -- be other, anything else is probably wrong, but do it anyway
    if (R.D = 0) and (C_descriptor.If != other) then
13.    ooc_jen <- po_len
        if C_descriptor.S = 0 then
14.    ooc_dmr <- set
        endif
15.    ooc_said <- C
        ooc_cfg.operate <- operate
    endif

    -- JC setup:
    -- if there is a register set targeted at JC
    if (D.R = 1) and (C_descriptor.topnibble = 0x9) then
16.    @C_descriptor <- C
    endif
17.    jc_cfg.instruction <- I.opcode
    jc_cfg.decode <- 1
    jc_cfg.operate <- 1

    -- Result Organiser:
    if R_descriptor.S = 0 then
18.    ro_dmr <- set
    endif
    -- if there is a register write targeted at RO
19.    ro_sa <- R
    if (D.R = 1) and (C_descriptor.topnibble = 0xB) then
20.    @C_descriptor <- C
    endif
21.    ro_cfg.mode <- jpeg
    ro_cfg.chan <- I.M+I.A
    ro_cfg.upsample <- I.S
    ro_cfg.cut <- I.C
    ro_cfg.llmt <- I.T

```

A.4.2 Compression

Notes

1. The following registers should be set prior to this instruction being executed:
 - po_idr: output image dimensions register
 - je_rmt: restart marker interval
 - ro_cut: output cut register
 - ro_lmt: output limit register

```

-- if there is a register set targeted at anything other than
-- the JC, PO, RO or DCC
if (D.R = 1) and (C_descriptor.topnibble != 0x5, 0x6, 0x9 or 0xB) then
0.    @C_descriptor <- C
    endif

-- Pixel Organiser:

```

```

-- Operand A is the source compressed data, and should be a
-- byte stream
if (R = 0) then
1.   po_len <- 0xC000:1:len
endif
if A_descriptor.S = 0 then
2.   po_dmr <- set
endif
3.   po_said <- A
   -- if there is a register set targeted at PD
   if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
4.     @C_descriptor <- C
   endif
5.   po_cfg.mode <- jpeg
   po_cfg.dst <- jc
   po_cfg.ss <- 1.5           -- subsampling
   po_cfg.chan <- 1.M:1.4
   po_cfg.F <- 1.F

   -- DCC setup:
   if B_descriptor.H = other then
6.     dcc_addr <- B
   -- if there is a register set targeted at DCC
   if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
7.     @C_descriptor <- C
   endif
8.     dcc_cfg2.cache_miss_inst <- B_descriptor.C
     dcc_cfg2.mode <- JPEG encode
   end if

   -- Operand Organiser B:
   -- Operand B should be other, anything else is probably
   -- wrong, but do it anyway
   if B_descriptor.H != other then
9.     oob_len <- po_len
     if B_descriptor.S = 0 then
10.      oob_dmr <- set
     endif
11.    oob_said <- B
12.    oob_cfg.operate <- operate
   endif

   -- Operand Organiser C:
   -- Operand C may be a register to set. If it isn't it should
   -- be other, anything else is probably wrong, but do it
   -- anyway
   if (R.D = 0) and (C_descriptor.H != other) then
13.     ooc_len <- po_len
     if C_descriptor.S = 0 then
14.      ooc_dmr <- set
     endif
15.    ooc_said <- C
16.    ooc_cfg.operate <- operate
   endif

   -- JPEG coder:
   -- if there is a register set targeted at JC
   if (D.R = 1) and (C_descriptor.topnibble = 0x9) then

```

```

17.   @C_descriptor <- C
      endif
18.   jc_cfg.instruction <- I.opcode
      jc_cfg.decode < 1
      jc_cfg.operate <- 1

      -- Result Organiser:
      if R_descriptor.S = 0 then
19.     ro_dmr <- set
      endif
20.   ro_sa <- R
      -- If there is a register write targeted at RO
      if (D.R = 1) and (C_descriptor.topnibble = 0xB) then
21.     @C_descriptor <- C
      endif
22.   ro_cfg.mode <- sequential
      ro_cfg.cut <- I.C
      ro_cfg.limit <- I.T

```

A.5 Data Coding

Notes

1. All data coding operations, whether compressing or decompressing are dealt with in the same way. Setting up for these operations looks very similar to that for JPEG.
2. Possible encoding operations are:
 - huffman encode
 - predictive encode
3. Possible decoding operations are:
 - fast huffman decode
 - slow huffman decode
 - packbits decoding (version A)
 - packbits decoding (version B)
 - predictive decode
4. Operand C may be a register to set
5. The following registers may be required to be set prior to this instruction being executed:
 - ro_cut: output cut register
 - ro_lmt: output limit register

```

-- if there is a register set targeted at anything other than
-- the JC, PO, RO or DCC
if (D.R = 1) and (C_descriptor.topnibble != 0x5, 0x6, 0x9 or 0xB) then
0.   @C_descriptor <- C
      endif

-- Pixel Organiser:
-- Operand A is the source data
if I.R = 0 then
1.   po_len <- 0x0000:ilen
      endif
      if A_descriptor.S = 0 then
2.     po_dmr <- set
      endif
3.   po_said <- A
      -- if there is a register set targeted at PO
      if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
4.     @C_descriptor <- C
      endif

```



```

endif
5. po_cfg.mode <- sequential
   po_cfg.dst <- jc

   -- DCC setup:
   if B_descriptor.if = other then
6.   dcc_addr <- B
      if there is a register set targeted at DCC
      if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
7.   @C_descriptor <- C
      endif
8.   dcc_cfg2.cache_miss_inst <- B_descriptor.C
      dcc_cfg2.mode <- en/dc coding
   else
      -- if there is a register set targeted at DCC
      if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
        @C_descriptor <- C
      endif
      dcc_cfg2.cache_miss_inst <- B_descriptor.C
   endif

   -- Operand Organiser B:
   -- Operand B should be other, anything else is probably
   -- wrong, but do it anyway
   if B_descriptor.if != other then
9.   oob_len <- po_len
      if B_descriptor.S = 0 then
10.  oob_dmr <- set
      endif
11.  oob_said <- B
12.  oob_cfg.operate <- operate
   endif

   -- Operand Organiser C:
   -- Operand C may be a register to set. If it isn't it should
   -- be other, anything else is probably wrong, but do it
   -- anyway
   if (R.D = 0) and (C_descriptor.if != other) then
13.  ooc_len <- po_len
      if C_descriptor.S = 0 then
14.  ooc_dmr <- set
      endif
15.  ooc_said <- C
16.  ooc_cfg.operate <- operate
   endif

   -- JPEG Coder:
   -- if there is a register set targeted at JC
   if (D.R = 1) and (C_descriptor.topnibble = 0x9) then
17.  @C_descriptor <- C
   endif
18.  jc_cfg.instruction <- I.opcode
      jc_cfg.decode <- 1
      jc_cfg.operate <- 1

   -- Result Organiser:
   if R_descriptor.S = 0 then
19.  ro_dmr <- set

```

```

endif
20. ro_sa <- R
   -- if there is a register write targeted at R0
   if (D.R = 1) and (C_descriptor.topnibble = 0xB) then
21.   C_descriptor <- C
   endif
22. ro_cfg.mode <- sequential
   ro_cfg.cnt <- LC
   ro_cfg.limit <- LT

```

A.6 Transformations and Convolutions

Notes

1. Opcode = 0x4 (convolutions) and 0x5 (transformations)
2. the coprocessor performs an operation which is a superset of what is required for each of Image Transformations and Convolutions. As far as the coprocessor is concerned the only difference between a transformation and a convolution is that for a transformation, the kernel step size (horizontally and vertically) is the size of the kernel, whereas for a convolution, the step size is one source pixel.
3. Options are:
 - interpolating or snapping-to-closest pixels
 - pixel accumulation (i.e. a kernel) or not
 - pre-multiplication or not of the source pixels
 - clamp, wrap or absolute value for determining final result
4. Note that transformations and convolutions don't work in place, i.e. if the source and destination pointers are the same it will break.

```

-- Pixel Organisier:
-- Operand A is the kernel descriptor and the PO delivers
-- kernel co-efficients to the MDP. This is coded as
-- "other"
-- L = 1 long format
--   0 short format
if LR = 0 then
1.   po_len <- 0x0000:llen
endif
if A_descriptor.S = 0 then
2.   po_dmr <- set
endif
3.   po_said <- A
4.   po_cfg.mode <- convolution/transformation
   po_cfg.dst <- mdp

-- DCC setup:
-- the implied data type here is the source image
if B_descriptor.lf = other then
5.   dcc_cfg2.cache_miss_inst <- B_descriptor.C
   if LS = 0 then
       dcc_cfg2.mode <- 64 bit mode
   else
       dcc_cfg2.mode <- random mode
   endif
else
   dcc_cfg2.cache_miss_inst <- B_descriptor.C
endif

```

```

-- Operand Organiser B:
-- Operand B is a pointer to source image (other), anything
else is probably wrong, but do it anyway
if B_descriptor.if != other then
6.   oob_len <- po_len
   if B_descriptor.S = 0 then
7.     oob_dmr <- set
   endif
8.   oob_said <- B
9.   oob_cfg.operate <- operate
endif

-- Operand Organiser C:
-- operand C descriptor is borrowed by the r.c field
-- so OOC is not set up

-- Main Data Path:
-- kernel descriptor is delivered to MDP via PO
10.  mdp_len <- po_len
11.  mdp_bm.rows <- C_descriptor.r
   mdp_bm.cols <- C_descriptor.c
12.  mdp_cfg.instruction <- I_opcode
   mdp_cfg.long_krn1 <- A_descriptor.L
   mdp_cfg.decode <- 1
   mdp_cfg.operate <- 1

-- Result Organiser:
-- result will be pixels or part thereof
if R_descriptor.S = 0 then
13.  ro_dmr <- set
endif
14.  ro_sa <- R
15.  ro_cfg.mode <- sequential

```

A.7 Matrix Multiplication

Notes

1. Opcode = 0x3
2. Options are:
 - pre-multiplication or not of the source pixels
 - clamp, wrap or absolute value for determining final result
 - Operand C may be a register write

```

-- if there is a register set targeted at anything other than
-- the MDP, PO, RO or DCC
if (D.R = 1) and (C_descriptor.topnibble != 0x5, 0x6, 0xA or 0xB) then
0.   @C_descriptor <- C
endif

-- Pixel Organiser:
-- Operand A is the source pixels. only makes sense to have
whole pixels, anything else is probably wrong
if I.R = 0 then
1.   po_len <- 0x0000:llen
endif
if A_descriptor.S = 0 then

```

```

2.   po_dmr <- set
   endif
3.   po_said <- A
   -- if there is a register set targeted at PO
   if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
4.     @C_descriptor <- C
   endif
5.   po_cfg.mode <- sequential
   po_cfg.dst <- mdp

   -- DCC setup:
   -- the implied other data type is a matrix of coefficients
   if B_descriptor.If = other then
6.     dcc_addr <- B
   endif
   -- if there is a register write targeted at DCC
   if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
7.     @C_descriptor <- C
   endif
   if B_descriptor.If = other
8.     dcc_cfg2.cache_miss_inst <- B_descriptor.C
     dcc_cfg2.mode <- matrix multiply
   else
     dcc_cfg2.cache_miss_inst <- B_descriptor.C
   endif

   -- Operand Organiser B:
   -- Operand B is the matrix co-efficients (other),
   -- anything else is probably wrong, but do it anyway
   if B_descriptor.If != other then
9.     oob_len <- po_len
     if B_descriptor.S = 0 then
10.      oob_dmr <- set
     endif
11.    oob_said <- B
12.    oob_cfg.operate <- operate
   endif

   -- Operand Organiser C:
   -- Operand C may be a register to set. If it isn't it should
   -- be other, anything else is probably wrong but do it anyway
   if (R.D = 0) and (C_descriptor.If != other) then
13.     ooc_len <- po_len
     if C_descriptor.S = 0 then
14.      ooc_dmr <- set
     endif
15.    ooc_said <- C
16.    ooc_cfg.operate <- operate
   endif

   -- MDP setup:
   -- if there is a register set targeted at MDP
   if (D.R = 1) and (C_descriptor.topnibble = 0xA) then
17.     @C_descriptor <- C
   endif
18.   mdp_cfg.instruction <- 1.opcode
   mdp_cfg.decode <- 1
   mdp_cfg.operate <- 1

```

```

-- Result Organiser:
if R_descriptor.S = 0 then
19.   ro_dmr <- set
endif
20.   ro_sa <- R
    -- if there is a register write targeted at RO
    if (D.R = 1) and (C_descriptor.topnibble = 0xB) then
21.     @C_descriptor <- C
    endif
22.   ro_cfg.mode <- sequential

```

A.8 Halftoning

Notes

1. Opcode = 0x7
2. Only option is the number of levels of halftoning
3. can be done on pixels or bytes as long as the halftone screen is appropriately unmeshed or meshed.

```

-- Pixel Organiser:
-- A operand is the source pixels or bytes
-- PO operates in conventional sequential mode
if LR = 0 then
1.   po_len <- 0x0000:llen
endif
if A_descriptor.S = 0 then
2.   po_dmr <- set
endif
3.   po_said <- A
4.   po_cfg.mode <- sequential
   po_cfg.dst <- mdp

-- DCC setup:
-- no implied other data type
-- if there is a register write targeted at DCC
if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
5.   @C_descriptor <- C
endif
if B_descriptor.C = 0 then
6.   dcc_cfg2.cache_miss_inst <- B_descriptor.C
endif

-- Operand Organiser B:
-- Operand B is the halftone screen, and should be a tile
-- anything else is probably wrong. There is no sensible
-- implied "other" data type for halftoning.
if B_descriptor.H != other then
7.   oob_len <- po_len
   if B_descriptor.S = 0 then
8.     oob_dmr <- set
   endif
9.   oob_said <- B
   if B_descriptor.what = tile
10.    oob_tile <- C
   endif

```

```

11.    oob_cfg.operate <- operate
      endif

      -- Operand Organiser C:
      -- Operand C word is borrowed by the tiled Operand B.
      -- The C_descriptor is completely ignored
      -- OOC is in random mode (default) and wont be touched

      -- Main Data Path:
12.    mdp_bm.level <- 1.levels
13.    mdp_cfg.instruction <- 1.opcode
      mdp_cfg.decode <- 1
      mdp_cfg.operate <- 1

      -- Result Organiser:
      if R_descriptor.S = 0 then
14.    ro_dmr <- set
      endif
15.    ro_sa <- R
16.    ro_cfg.mode <- sequential

```

A.9 Memory Copy

Notes

1. Opcode = 0x9
2. This instruction uses two quite distinct mechanisms to achieve a memory copy operation:
 - A General Data Transfer utilises the normal data flow path through the coprocessor and so can utilise the various functions associated with the data manipulation units in the PO and RO.
 - A Peripheral DMA utilises a direct connection between the PIC and the LMC. This means that no data manipulation can be performed, and that this operation may be overlapped with subsequent instructions.

A.9.1 General Data Transfer

```

-- Pixel Organiser:
-- A operand is the source data
-- PO operates in conventional sequential mode
if I.R = 0 then
1.    po_len <- 0x0000:1.len
    endif
    if A_descriptor.S = 0 then
2.    po_dmr <- B
    end if
3.    po_said <- A
4.    po_cfg.mode <- sequential
    po_cfg.dst <- jc

-- Operand Organiser B:
-- Operand B word represents a value to put in po_dmr

-- Operand Organiser C:
-- Operand C word represents the value to put in the ro_dmr

-- JPEG Coder:

```

```

5.  jc_cfg.instruction <- I.opcode
    jc_cfg.lbo <- D.bo
    jc_cfg.oho <- I.obo
    jc_cfg.decode <- 1
    jc_cfg.operate <- 1

    Result Organiser:
    if R_descriptor.S = 0 then
6.      ro_dmr <- C
    end
7.      ro_sa <- R
8.      ro_cfg.mode <- sequential

```

A.9.2 Peripheral DMA

Notes

1. May be overlapped or not. This is dealt with by the IC
2. Operand C may be a register to set
3. This instruction is different from other "activity" instructions in that the PIC is the module that drives the data.

```

-- only setup required is for the PIC:
if I.S = 1 then
1.      picabus_addr <- R
    else
        picabus_addr <- A
    endif
    -- if there is a register write targeted anywhere
    if D.R = 1 then
2.      @C_descriptor <- C
    endif
    if I.R = 0 then
3.      picabus_cfg.ab_count <- 0x00:I.length
    endif
    picabus_cfg.ab_byte_en <- I.byte
    picabus_cfg.ab_type <- I.R
    picabus_cfg.start <- 1
    -- Pixel Organiser:
    -- Operand A is the source address given to the PIC
    -- The PO is not setup

    -- Operand Organiser B:
    -- OOB is not setup

    -- Operand Organiser C:
    -- OOC is not setup

    -- Result Organiser:
    -- The Result word represents the target address
    -- RO is not set up

```

A.10 PhotoCD Decompression

This family of instructions consists of three different operations: horizontal interpolation, vertical interpolation and residual merging. As it happens, the setup for Vertical interpolation and the setup for residual merging are identical.

Opcode for all these instructions is 0x9

A.10.1 Horizontal Interpolation

Notes

1. can operate on pixels or bytes
2. this is a one operand instruction so Operand C may be a register to set

```
-- if there is a register set targeted at anything other than
-- the MDP, PO or RO
if (D.R = 1) and (C_descriptor.topnibble != 0x6, 0xA or 0xB) then
0.   @C_descriptor <- C
endif

-- Pixel Organiser:
if L.R = 0 then
1.   po_len <- 0x0000:1:len
endif
if A_descriptor.S = 0 then
2.   po_dmr <- set
endif
po_said <- A
-- if there is a register set targeted at PO
if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
4.   @C_descriptor <- C
endif
5.   po_cfg.mode <- sequential
   po_cfg.dst <- mdp

-- Operand Organiser B:
-- Operand B word is completely ignored
-- The B_descriptor is completely ignored

-- Operand Organiser C:
-- Operand C may be a register to set up

-- Main Data Path:
-- if there is a register set targeted at MDP
if (D.R = 1) and (C_descriptor.topnibble = 0xA) then
6.   @C_descriptor <- C
endif
7.   mdp_cfg.instruction <- 1:opcode
   if A_descriptor.H /= packed bytes then-- tell mdp about size
       mdp_cfg.word_input <- 1
   else
       mdp_cfg.word_input <- 0
   endif
   mdp_cfg.decode <- 1
   mdp_cfg.operate <- 1

-- Result Organiser:
```



```

      if Rdescriptor.S = 0 then
8.      ro_dmr <- set
      endif
9.      ro_sa <- R
10.     ro_cfg.mode <- sequential

```

A.10.2 Vertical Interpolation and Residual Merging

Notes

1. The setup for Vertical Interpolation and Residual Merging is identical.
2. Can operate on either bytes or pixels
3. This is a two operand instruction so Operand C may be a register set

```

-- if there is a register set targeted at anything other than
-- the MDP, PO, OOB or RO
if (D.R = 1) and (C_descriptor.topnibble != 0x6, 0x7, 0xA or 0xB) then
0.     @C_descriptor <- C
    endif

-- Pixel Organiser:
if I.R = 0 then
1.     po_len <- 0x0000:llen
    endif
if A_descriptor.S = 0 then
2.     po_dmr <- set
    endif
3.     po_said <- A
    -- if there is a register set targeted at PO
    if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
4.         @C_descriptor <- C
        endif
5.     po_cfg.mode <- sequential
    po_cfg.dst <- mdp

-- DCC setup:
-- no implied other data type
-- if there is a register write targeted at DCC
if (D.R = 1) and (C_descriptor.topnibble = 0x5) then
6.     @C_descriptor <- C
    endif
    if B_descriptor.C = 0 then
        doc_cfg2.cache_miss_inst <- B_descriptor.C
    endif

-- Operand Organiser B:
-- Operand B is the residuals or one operand for
-- interpolation, anything else is probably wrong, but
-- do it anyway
-- There is no sensible implied "other" data type for
-- this.
if B_descriptor.If != other then
7.     oob_len <- po_len
        if B_descriptor.S = 0 then
8.             oob_dmr <- set
        endif
9.     oob_said <- B

```

```

endif
-- if there is a register set targeted at ODB
  if (D.R = 1) and (C_descriptor.topnibble = 0x6) then
10.    @C_descriptor <- C
  endif
11.    odb_cfg.operate <- operate
endif

-- Operand Organiser C:
-- may be a register to set

-- Main Data Path:
-- if there is a register set targeted at MDP
  if (D.R = 1) and (C_descriptor.topnibble = 0xA) then
12.    @C_descriptor <- C
  endif
13.  mdp_cfg.instruction <- I.opcode
  if A_descriptor.if /= packed_bytes then-- tell mdp about size
    mdp_cfg.word_input <- 1
  else
    mdp_cfg.word_input <- 0
  endif
  mdp_cfg.decode <- 1
  mdp_cfg.operate <- 1

-- Result Organiser:
  if R_descriptor.S = 0 then
14.    ro_dmr <- set
  endif
15.  ro_sa <- R
16.  ro_cfg.mode <- 0
-- sequential

```

A.11 Control Instructions

Notes

1. Control Instructions consist of two classes of operations: Flow Control Instructions and Internal Access Instructions.

A.11.1 Flow Control

Notes

1. Opcode = 0xB
2. Flow Control Instructions currently consist of Jumps of various kinds and Waits of various kinds.
3. No explicit setup is done in the coprocessor, and this instruction is not an "activity" instruction, ie no the coprocessor submodules other than the instruction controller actually do anything.
4. Operand C may be a register to set.

```

-- only thing to do is:
if D.R = 1 then
0.    @C_descriptor <- C
endif
-- Pixel Organiser:

```

- no setup associated with Operand A
- Operand Organiser B:
- no setup associated with Operand B
- Operand Organiser C:
- no setup associated with Operand C
- Result Organiser:
- no setup associated with the Result

A.11.2 Internal Access: Read

Notes

1. Opcode = 0xA
2. A "read" is a transfer of data out of the coprocessor
3. The only module in the coprocessor that actually does anything for this is the RO

- Pixel Organiser:
 - no setup associated with Operand A
 - Operand Organiser B:
 - no setup associated with Operand B
 - Operand Organiser C:
 - no setup associated with Operand C
 - Result Organiser
1. ro_subst <- Cbus(A)
 2. ro_sa <- B
 3. ro_cfg.mode <- Cbus_data

A.11.3 Internal Access: Write

Notes

1. Opcode = 0xA
2. A "write" is a transfer of data into the coprocessor.
3. This instruction is not an "activity" instruction as no modules other than the IC actually do anything.

1. Cbus(A) <- B
- Pixel Organiser:
 - no setup associated with Operand A
 - Operand Organiser B:
 - no setup associated with Operand B
 - Operand Organiser C:
 - no setup associated with Operand C
 - Result Organiser:
 - no setup associated with the Result

A.12 Reserved Instructions

Notes

1. Opcodes 0x0, 0xE and 0xF are "reserved".
2. A "reserved" instruction will generate a (maskable) error.
3. These "reserved" opcodes are intended to be used for other instructions that may be added in later revisions of the coprocessor.

Appendix B: Registers

1.1 Registers and Tables

This Section describes the coprocessor's registers. These registers can be modified in one of three ways:

1. Specific the coprocessor instructions exist for reading and writing registers. By using these instructions these registers can be read or written using initiator PCIbus cycles, or by transactions on the Generic Interface, or to/from any memory that may be connected to the Local Memory Interface.
2. Many registers change value as a side effect of instruction execution. The primary mechanism by which the coprocessor configures itself for instruction execution is by setting various registers to reflect the currently executing instruction. At the end of the instruction, the registers reflect the state of the coprocessor at that time.

Most typical operations can be completely specified and set up by the one instruction. In some cases, it is necessary to set some registers immediately prior to executing the instruction.

Semantics of "reserved" register bits

Any registers or parts thereof that are "reserved" have the following semantics:

- a write to a reserved location will succeed and the data will be discarded
- a read from a reserved location will succeed and will result in undefined value

All unspecified registers and register fields are reserved.

1.1.1 Register Taxonomy

The registers in the coprocessor are classified into types based on their behavior as described in this section. In these descriptions:

- externally = external to the module, i.e. via the Cbus interface, i.e. via target mode PCI, by the Instruction controller or via the external Cbus interface. Note that the registers can't be set via bit-set (Type C) mode from the PCIbus.
- internally = internally to the module

Status Register

A status register is read-only externally and read/write internally.

Config1 Register

A Config1 register is read/write externally and read only internally.

A Config1 register does not support Type C Cbus operations (ie does not support bit set mode) and is used for registers that hold byte (or bigger) type configuration information such as address values.

Config2 Register

A Config2 register is also read/write externally and read only internally.

A Config2 register supports Type C Cbus operations (ie supports bit set mode) and is used for registers that hold configuration information that may need to be set on a bit by bit basis.

Control1 Register

A Control1 register is readable/writable both internally and externally.

A Control1 register does not support Type C Cbus operations (ie does not support bit set mode) and is used for registers that hold byte (or bigger) type control information such as address values.

Control2 Register

A Control2 register is readable/writable both internally and externally.

A Control2 register supports Type C Cbus operations (ie supports bit set mode) and is used for registers that hold control information that may need to be set on a bit by bit basis.

Interrupt Register

The bits within an Interrupt register are settable to 1 internally and resettable to 0 externally by writing a 1 to that bit.

Each of the module interrupt/error registers are of this type. Each module's interrupt/error register consists of three fields:

[7:0] represents any *error* conditions generated by the module

[23:8] represents any *exception* conditions generated by the module

[31:24] represents any *interrupt* conditions generated by the module.

1.1.2 Register Map

Table 1.1 shows the coprocessor's registers. The "number" in Table 1.1 is the number of the register rather than a byte address.

Table 1.1 the coprocessor registers

number	name	type	description	see page
External Interface Controller Registers				page 26
0x00	eic_cfg	Config2	Configuration	
0x01	eic_stat	Status	Status	
0x02	eic_err_int	Interrupt	Error and Interrupt Status	
0x03	eic_err_int_en	Config2	Error and Interrupt Enable	
0x04	eic_test	Config2	Test modes	
0x05	eic_gen_pob	Config2	Generic bus programmable output bits.	
0x06	eic_high_addr	Config1	Dual address cycle offset	
0x07				
0x08	eic_wtlb_v	Control2	Virtual address and operation bits for TLB Invalidate/Write	
0x09	eic_wtlb_p	Config2	Physical address and control bits for TLB Write	
0x0A	eic_mmu_v	Status	Most recent MMU virtual address translated, and current LRU location.	
0x0B	eic_mmu_p	Status	Most recent page table physical address fetched by MMU.	
0x0C	eic_ip_addr	Status	Physical address for most recent IBus access to the PCI Bus.	
0x0D	eic_rp_addr	Status	Physical address for most recent RBus access to the PCI Bus.	

Table 1.1 the coprocessor registers

number	name	type	description	see page
0x0E	eic_ig_addr	Status	Address for most recent IBus access to the Generic Bus.	
0x0F	eic_rg_data	Status	Address for most recent RBus access to the Generic Bus.	
Local Memory Controller Registers				page 32
0x10	lmi_cfg	Control2	General configuration register	
0x11	lmi_sts	Status	General status register	
0x12	lmi_err_int	Interrupt	Error and interrupt status register	
0x13	lmi_err_int_en	Control2	Error and interrupt enable register	
0x14	lmi_dcfg	Control2	DRAM configuration register	
0x15	lmi_mode	Control2	SDRAM mode register	
0x16				
0x17				
0x18				
0x19				
0x1A				
0x1B				
0x1C				
0x1D				
0x1E				
0x1F				
Peripheral Interface Controller Registers				page 36
0x20	pic_cfg	Config2	Configuration	
0x21	pic_stat	Status	Status	
0x22	pic_err_int	Interrupt	Interrupt/Error Status	
0x23	pic_err_int_en	Config2	Interrupt/Error Enable	
0x24	pic_abus_cfg	Control2	Configuration and control for ABus	
0x25	pic_abus_addr	Config1	the coprocessor address for ABus transfer	
0x26	pic_cent_cfg	Control2	Configuration and control for Centronics mode.	
0x27	pic_cent_dir	Config2	Centronics pin direct control register	
0x28	pic_reverse_cfg	Control2	Configuration and control for reverse (input) data transfers	
0x29				
0x2A	pic_timer0	Config1	Initial data timer value.	
0x2B	pic_timer1	Config1	Subsequent data timer value.	
0x2C				
0x2D				
0x2E				
0x2F				
Miscellaneous Module Registers				page 8
0x30	mmu_cfg	Config2	Configuration Register	

Table 1.1 the coprocessor registers

number	name	type	description	see page
0x31	mm_stat	Status	Status Register	
0x32	mm_err_int	Interrupt	Error and Interrupt Register	
0x33	mm_err_int_en	Config2	Error and Interrupt Masks	
0x34	mm_gcfg	Config2	Global Configuration Register	
0x35	mm_diag	Config	Diagnostic Configuration Register	
0x36	mm_rst	Config*	Global Reset Register	
0x37	mm_gerr	Config2	Global Error Register	
0x38	mm_gexp	Config2	Global Exception Register	
0x39	mm_gint	Config2	Global Interrupt Register	
0x3A	mm_active	status	Global Active signals	
0x3B				
0x3C				
0x3D				
0x3E				
0x3F				
Instruction Controller Registers				page 15
0x40	ic_cfg	Config2	Configuration Register	
0x41	ic_stat	Status/ Interrupt	Status Register	
0x42	ic_err_int	Interrupt	Error and Interrupt Register (write to clear error and interrupt)	
0x43	ic_err_int_en	Config2	Error and Interrupt Enable Register	
0x44	ic_ipa	Control 1	A stream Instruction Pointer	
0x45	ic_tda	Config1	A stream Todo Register	
0x46	ic_fna	Control 1	A stream Finished Register	
0x47	ic_inta	Config1	A stream Interrupt Register	
0x48	ic_loa	Status	A stream Last Overlapped Instruction Sequence number	
0x49	ic_ipb	Control 1	B stream Instruction Pointer	
0x4A	ic_tdb	Config1	B stream Todo Register	
0x4B	ic_fnb	Control 1	B stream Finished Register	
0x4C	ic_intb	Config1	B stream Interrupt Register	
0x4D	ic_lob	Status	B stream Last Overlapped Instruction Sequence number	
0x4E	ic_sema	Status	A stream Semaphore	
0x4F	ic_semb	Status	B stream Semaphore	
Data Cache Controller Registers				page 42
0x50	dcc_cfg1	config2	DCC configuration 1 register	
0x51	dcc_stat	status	state machine status bits	
0x52	dcc_err_int	status	DCC error status register	
0x53	dcc_err_int_en	control 1	DCC error interrupt enable bits	
0x54	dcc_cfg2	control2	DCC configuration 2 register	

Table 1.1 the coprocessor registers

number	name	type	description	see page
0x55	ccc_addr	config1	Base address register for special address modes.	
0x56	dcc_lv0	control1	"valid" bit status for lines 0 to 31	
0x57	dcc_lv1	control1	"valid" bit status for lines 32 to 63	
0x58	dcc_lv2	control1	"valid" bit status for lines 64 to 95	
0x59	dcc_lv3	control1	"valid" bit status for lines 96 to 127	
0x5A	dcc_raddrb	status	Operand Organiser B request address	
0x5B	dcc_raddrc	status	Operand Organiser C request address	
0x5C	dcc_test	control1	DCC test register	
0x5D				
0x5E				
0x5F				
Pixel Organiser Registers				page 50
0x60	po_cfg	Config2	Configuration Register	
0x61	po_stat	Status	Status Register	
0x62	po_err_int	Interrupt	Error/Interrupt Status Register	
0x63	po_err_int_en	Config2	Error/Interrupt Enable Register	
0x64	po_dmr	Config2	Data Manipulation Register	
0x65	po_subst	Config2	Substitution Value Register	
0x66	po_cdp	Status	Current Data Pointer	
0x67	po_len	Control1	Length Register	
0x68	po_said	Control1	the coprocessor Address or Immediate Data	
0x69	po_idr	Control2	Image Dimensions Register	
0x6A	po_muv_valid	Control2	MUY valid bits	
0x6B	po_muv	Config1	Base address of MUV RAM	
0x6C				
0x6D				
0x6E				
0x6F				
Operand Organiser B Registers				page 46
0x70	oob_cfg	Config2	Configuration Register	
0x71	oob_stat	Status	Status Register	
0x72	oob_err_int	Interrupt	Error/Interrupt Register	
0x73	oob_err_int_en	Config2	Error/Interrupt Enable Register	
0x74	oob_dmr	Config2	Data Manipulation Register	
0x75	oob_subst	Config2	Substitution Value Register	
0x76	oob_cdp	Status	Current Data Pointer	
0x77	oob_len	Control1	Input Length Register	
0x78	oob_said	Control1	Operand the coprocessor Address	
0x79	oob_tile	Control1	Tiling length/offset Register	
0x7A				

Table 1.1 the coprocessor registers

number	name	type	description	see page
0x7B				
0x7C				
0x7D				
0x7E				
0x7F				
Operand Organiser C Registers				page 46
0x80	ooc_cfg	Config2	Configuration Register	
0x81	ooc_stat	Status	Status Register	
0x82	ooc_err_int	Interrupt	Error/Interrupt Register	
0x83	ooc_err_int_en	Config2	Error/Interrupt Enable Register	
0x84	ooc_dmr	Config2	Data Manipulation Register	
0x85	ooc_subst	Config2	Substitution Value Register	
0x86	ooc_cdp	Status	Current Data Pointer	
0x87	ooc_len	Control1	Input Length Register	
0x88	ooc_said	Control1	Operand the coprocessor Address	
0x89	ooc_tile	Control1	Tiling length/offset Register	
0x8A				
0x8B				
0x8C				
0x8D				
0x8E				
0x8F				
JPEG Coder Registers				page 63
0x90	jc_cfg	Config2	configuration	
0x91	jc_stat	Status	status	
0x92	jc_err_int	Interrupt	error and interrupt status register	
0x93	jc_err_int_en	Config2	error and interrupt enable register	
0x94	jc_rsi	Config1	rethe coprocessor interval	
0x95	jc_decode	Control2	decode of current instruction	
0x96	jc_res	Control1	residual value	
0x97	jc_table_sel	Control2	table selection from decoded instruction	
0x98				
0x99				
0x9A				
0x9B				
0x9C				
0x9D				
0x9E				
0x9F				
Main Data Path Registers				page 54
0xA0	mdp_cfg	Config2	configuration	

Table 1.1 the coprocessor registers

number	name	type	description	see page
0xA1	mdp_stat	Status	status	
0xA2	mdp_err_int	Interrupt	error/interrupt	
0xA3	mdp_err_int_en	Config2	error/interrupt enable	
0xA4	mdp_test	Config2	test modes	
0xA5	mdp_op1	Control2	current operation 1	
0xA6	mdp_op2	Control2	current operation 2	
0xA7	mdp_por	Control1	offset for plus operator	
0xA8	mdp_bi	Control1	blend the coprocessor. offset to index table entry	
0xA9	mdp_bm	Control1	blend end or number of rows and columns in matrix, binary places, and number of levels in halftoning	
0xAA	mdp_len	Control1	Length of blend to produce	
0xAB				
0xAC				
0xAD				
0xAE				
0xAF				
Result Organiser Registers				page 68
0xB0	ro_cfg	Config2	Configuration Register	
0xB1	ro_stat	Status	Status Register	
0xB2	ro_err_int	Interrupt	Error/Interrupt Register	
0xB3	ro_err_int_en	Config2	Error/Interrupt Enable Register	
0xB4	ro_dmr	Config2	Data Manipulation Register	
0xB5	ro_subst	Config1	Substitution Value Register	
0xB6	ro_cdp	Status	Current Data Pointer	
0xB7	ro_len	Status	Output Length Register	
0xB8	ro_sa	Config1	the coprocessor Address	
0xB9	ro_idr	Config1	Image Dimensions Register	
0xBA				
0xBB	ro_vbase	Config1	the coprocessor Virtual Base Address	
0xBC	ro_cut	Config1	Output Cut Register	
0xBD	ro_lmt	Config1	Output Length Limit	
0xBE				
0xBF				
PCibus Configuration Space alias				
0xC0- 0xCD, 0xCF			A read only copy of PCI configuration space registers 0x0 to 0xD and 0xF.	page 73
0xCE	pci_external_cfg	Status	32-bit field downloaded at reset from an external serial ROM. Has no influence on the coprocessor's operation.	

Table 1.1 the coprocessor registers

number	name	type	description	See page
Input Interface Switch Registers				page 23
0xD0	iis_cfg	Cpnfig2	Configuration Register	
0xD1	iis_stat	Status	Status Register	
0xD2	iis_err_int	Interrupt	Interrupt/Error Status Register	
0xD3	iis_err_int_en	Config2	Interrupt/Error Enable Register	
0xD4	iis_ic_addr	Status	Input address from IC	
0xD5	iis_dcc_addr	Status	Input address from DCC	
0xD6	iis_po_addr	Status	Input address from PO	
0xD7	iis_burst	Status	Burst Length from PO, DCC & IC	
0xD8	iis_base_addr	Config1	Base address of the coprocessor memory object in host memory map	
0xD9	iis_test	Config1	Test mode register	
0xDA				
0xDB				
0xDC				
0xDD				
0xDE				
0xDF				
0xE0-0xFF			not used	

1.1.3 Register Definitions

Miscellaneous Module Registers

a. mm_cfg

Table 1.2 mm_cfg Register Fields

Field	Name	Description
1	E	0=disabled 1=enabled
2	B	0=not bypassed 1=bypassed

b. mm_stat

Table 1.3 mm_stat Register Fields

Field	Name	Description
		Reserved

c. mm_err_int

Table 1.4 mm_err_int Register Fields

Field	Name	Description
		Reserved

d. mm_err_int_en

Table 1.5 mm_err_int_en Register Fields

Field	Name	Description
		Reserved

e. mm_gcfg

Table 1.6 mm_gcfg Register Fields

Field	Name	Description
0	use_pci_clk	(read only) value of the use_pci_clk signal 0=use clk_in to generate clk 1=use pci_clk to generate clk
1	slow_clk	(read only) divide source clock by 2 to generate clk 0=divide by 2 (default) 1=do not divide by 2
2	cbus_config	(read only) 0 = generic bus configured 1 = external cbus configured
31:1		Reserved

f. mm_diag

Table 1.7 mm_diag Register Fields

Field	Name	Description
1:0	diag_icfg[1:0]	program which diagnostics appear on the diag_i pin: 00= the coprocessor busy 01=new A stream instruction strobe 10=new B stream instruction strobe 11=new either stream instruction strobe
3:2	diag_dcfg[1:0]	program which diagnostics appear on the diag_d pins: 00=activity diagnostics: diag_d[0] = PCIbus master transaction diag_d[1] = generic bus transaction diag_d[2] = local memory transaction diag_d[3] = peripheral interface transaction diag_d[4] = non overlapped instruction active diag_d[5] = overlapped instruction active 01=instruction diagnostics: diag_d[3:0] = current instruction opcode diag_d[4] = current instruction stream diag_d[5] = unused 10=caching diagnostics: diag_d[0] = data cache hit diag_d[1] = servicing data cache miss diag_d[2] = TLB hit diag_d[3] = servicing TLB miss diag_d[4] = MUV hit diag_d[5] = servicing MUV miss 11=stall diagnostics: diag_d[0] = waiting on output from PO diag_d[1] = waiting on output from OOB diag_d[2] = waiting on output from OOC diag_d[3] = stall due to RO fifo full diag_d[4] = waiting on instruction fetch diag_d[5] = unused

g. mm_grst

Table 1.8 mm_grst Register Fields

Field	name	Description
15:0	msrst[15:0]	<p>Per-module software resets.</p> <p>Write 1 to appropriate bit to cause software reset. These bits are self clearing.</p> <p>bit 0 = External Interface Controller</p> <p>bit 1 = Local Memory Controller</p> <p>bit 2 = Peripheral Interface Controller</p> <p>bit 3 = Miscellaneous Module</p> <p>bit 4 = Instruction Controller</p> <p>bit 5 = Data Cache Controller</p> <p>bit 6 = Pixel Organiser</p> <p>bit 7 = Operand Organiser B</p> <p>bit 8 = Operand Organiser C</p> <p>bit 9 = JPEG Codec</p> <p>bit 10 = Main Data Path</p> <p>bit 11 = Result Organiser</p> <p>bit 12 = reserved</p> <p>bit 13 = Input Interface Switch</p> <p>bit 14 = reserved</p> <p>bit 15 = Software Reset <i>all</i> modules</p>

h. mm_gerr

Table 1.9 mm_gerr Register Fields

Field	Name	Description
31:16	err[15:0]	<p>Per-module error status bits:</p> <p>Note that these bits are read only. To clear the error condition, the appropriate module error bit should be cleared.</p> <p>0 = no error condition from this module</p> <p>1 = error condition from this module</p> <p>bit 0 = External Interface Controller</p> <p>bit 1 = Local Memory Controller</p> <p>bit 2 = Peripheral Interface Controller</p> <p>bit 3 = Miscellaneous Module</p> <p>bit 4 = Instruction Controller</p> <p>bit 5 = Data Cache Controller</p> <p>bit 6 = Pixel Organiser</p> <p>bit 7 = Operand Organiser B</p> <p>bit 8 = Operand Organiser C</p> <p>bit 9 = JPEG Coder</p> <p>bit 10 = Main Data Path</p> <p>bit 11 = Result Organiser</p> <p>bit 12 = reserved</p> <p>bit 13 = Input Interface Switch</p> <p>bit 14 = reserved</p> <p>bit 15 = error on <i>any</i> module</p>
15:10	err_en[15:0]	<p>Per-module error enables:</p> <p>0 = error conditions from this module are <i>not</i> translated into an interrupt of the host</p> <p>1 = error conditions from this module are translated into an interrupt of the host</p> <p>bit 0 = External Interface Controller</p> <p>bit 1 = Local Memory Controller</p> <p>bit 2 = Peripheral Interface Controller</p> <p>bit 3 = Miscellaneous Module</p> <p>bit 4 = Instruction Controller</p> <p>bit 5 = Data Cache Controller</p> <p>bit 6 = Pixel Organiser</p> <p>bit 7 = Operand Organiser B</p> <p>bit 8 = Operand Organiser C</p> <p>bit 9 = JPEG Coder</p> <p>bit 10 = Main Data Path</p> <p>bit 11 = Result Organiser</p> <p>bit 12 = reserved</p> <p>bit 13 = Input Interface Switch</p> <p>bit 14 = reserved</p> <p>bit 15 = interrupt on <i>any</i> module</p>

i. mm_gexp

Table 1.10 mm_gexp Register Fields

Field	Name	Description
31:16	exp[15:0]	<p>Per-module exception status bits:</p> <p>Note that these bits are read only. To clear the exception condition, the appropriate module exception bit should be cleared.</p> <p>0 = no exception condition from this module 1 = exception condition from this module</p> <p>bit 0 = External Interface Controller bit 1 = Local Memory Controller bit 2 = Peripheral Interface Controller bit 3 = Miscellaneous Module bit 4 = Instruction Controller bit 5 = Data Cache Controller bit 6 = Pixel Organiser bit 7 = Operand Organiser B bit 8 = Operand Organiser C bit 9 = JPEG Coder bit 10 = Main Data Path bit 11 = Result Organiser bit 12 = reserved bit 13 = Input Interface Switch bit 14 = reserved bit 15 = exception on <i>any</i> module</p>
15:0	exp_en[15:0]	<p>Per-module exception enables:</p> <p>0 = exception conditions from this module are <i>not</i> translated into an interrupt of the host 1 = exception conditions from this module are translated into an interrupt of the host</p> <p>bit 0 = External Interface Controller bit 1 = Local Memory Controller bit 2 = Peripheral Interface Controller bit 3 = Miscellaneous Module bit 4 = Instruction Controller bit 5 = Data Cache Controller bit 6 = Pixel Organiser bit 7 = Operand Organiser B bit 8 = Operand Organiser C bit 9 = JPEG Coder bit 10 = Main Data Path bit 11 = Result Organiser bit 12 = reserved bit 13 = Input Interface Switch bit 14 = reserved bit 15 = exception on <i>any</i> module</p>

j. mm_gint

Table 1.11 mm_gint Register Fields

Field	Name	Description
31:16	int[15:0]	<p>Per-module interrupt status bits:</p> <p>Note that these bits are read only. To clear the interrupt condition, the appropriate module interrupt bit should be cleared.</p> <p>0 = no interrupt condition from this module</p> <p>1 = interrupt condition from this module</p> <p>bit 0 = External Interface Controller</p> <p>bit 1 = Local Memory Controller</p> <p>bit 2 = Peripheral Interface Controller</p> <p>bit 3 = Miscellaneous Module</p> <p>bit 4 = Instruction Controller</p> <p>bit 5 = Data Cache Controller</p> <p>bit 6 = Pixel Organiser</p> <p>bit 7 = Operand Organiser B</p> <p>bit 8 = Operand Organiser C</p> <p>bit 9 = JPEG Coder</p> <p>bit 10 = Main Data Path</p> <p>bit 11 = Result Organiser</p> <p>bit 12 = reserved</p> <p>bit 13 = Input Interface Switch</p> <p>bit 14 = reserved</p> <p>bit 15 = interrupt on any module</p>
15:0	int_en[15:0]	<p>Per-module interrupt enables:</p> <p>0 = interrupt conditions from this module are <i>not</i> translated into an interrupt of the host</p> <p>1 = interrupt conditions from this module are translated into an interrupt of the host</p> <p>bit 0 = External Interface Controller</p> <p>bit 1 = Local Memory Controller</p> <p>bit 2 = Peripheral Interface Controller</p> <p>bit 3 = Miscellaneous Module</p> <p>bit 4 = Instruction Controller</p> <p>bit 5 = Data Cache Controller</p> <p>bit 6 = Pixel Organiser</p> <p>bit 7 = Operand Organiser B</p> <p>bit 8 = Operand Organiser C</p> <p>bit 9 = JPEG Coder</p> <p>bit 10 = Main Data Path</p> <p>bit 11 = Result Organiser</p> <p>bit 12 = reserved</p> <p>bit 13 = Input Interface Switch</p> <p>bit 14 = reserved</p> <p>bit 15 = interrupt on any module</p>

k. mm_active

Table 1.12 mm_active Register Fields

Field	Name	Description
15:0	mactive[15:0]	Per-module active signals (Read Only): These bits unlaunched bit 0 = External Interface Controller bit 1 = Local Memory Controller bit 2 = Peripheral Interface Controller bit 3 = Miscellaneous Module bit 4 = Instruction Controller bit 5 = Data Cache Controller bit 6 = Pixel Organiser bit 7 = Operand Organiser B bit 8 = Operand Organiser C bit 9 = JPEG Coder bit 10 = Main Data Path bit 11 = Result Organiser bit 12 = reserved bit 13 = Input Interface Switch bit 14 = reserved bit 15 = any module active
31:16		Reserved

Instruction Controller Registers

l. ic_cfg

The `ic_cfg` register is divided into three parts. The least significant byte contains global configuration information. The third least significant byte contains A stream configuration information and the most significant byte contains the B stream configuration information.

This register has a reset value of 0x00000000.

Table 1.13 ic_cfg Register Fields

Field	Name	Description
1	E	1=enabled 0=disabled
2	B	0=not bypassed 1=bypassed
3	S	0=not in self test mode 1=self test mode
[5:4]	pri[1:0]	instruction stream priority: 0=A high, B low 1=B high, A low 2=3=round robin

Table 1.13 ic_cfg Register Fields

Field	Name	Description
6	asr	Asynchronous stall reject: 0=not enabled 1=enabled
7	pd	Partial Decode enable: 0=not enabled 1=enabled
8	cbus_eic_dis	Disable requests from the EIC for Cbus cycles 0=do not disable 1=disable
9	cbus_mm_dis	Disable requests from the MM for Cbus cycles 0=do not disable 1=disable
10	cbus_int_dis	Disable requests from the IC for Cbus cycles 0=do not disable 1=disable
16	a_en	0=A stream disabled 1=A stream enabled
17	a_seq	0=A stream sequence numbers disabled 1=A stream sequence numbers enabled
18	a_prefetch	0=A stream prefetching disabled 1=A stream prefetching enabled
19	a_overlap_en	0=A stream overlapping disabled 1=A stream overlapping enabled
20	a_snc_pause	0 = A stream do not pause on "sequence number completed" interrupt 1 = A stream pause on "sequence number completed" interrupt
21	a_ic_pause	0 = A stream do not pause on "instruction completed" interrupt 1 = A stream pause on "instruction completed" interrupt
22	a_auto_prime	0=A stream "sequence number completed" interrupt not automatically primed 1=A stream "sequence number completed" interrupt automatically primed
24	b_en	0=B stream disabled 1=B stream enabled
25	b_seq	0=B stream sequence numbers disabled 1=B stream sequence numbers enabled
26	b_prefetch	0=B stream prefetching disabled 1=B stream prefetching enabled
27	b_overlap_en	0=B stream overlapping disabled 1=B stream overlapping enabled

Table 1.13 ic_cfg Register Fields

Field	Name	Description
28	b_snc_pause	0 = B stream do not pause on "sequence number completed" interrupt 1 = B stream pause on "sequence number completed" interrupt
29	b_ic_pause	0 = B stream do not pause on "instruction completed" interrupt 1 = B stream pause on "instruction completed" interrupt
30	b_auto_prime	0 = B stream "sequence number completed" interrupt not automatically primed 1 = B stream "sequence number completed" interrupt automatically primed

m. ic_stat

This register is split into four sections. The least significant byte contains the internal state of the IC. The second least significant byte contains the decoded result of the current instruction, and current and prefetched instruction stream. The second most significant byte contains all status information about the A stream, and the most significant byte contains information about the B stream.

The reset value of this register is 0x00000000.

Table 1.14 ic_stat Register Fields

Field	Name	Description
[3:0]	ec_state[3:0]	instruction execution state condition. 0 = idle 1 = fetching instruction 2 = waiting for overlapped instruction to finish 3 = decoding 4 = prefetching instruction 5 = wait for instruction to finish 6 = synchronise with external accesses prior to updating registers 7 = update state registers 8 = bad state
8	overlap	0 = overlapped instruction not in progress 1 = overlapped instruction in progress
9	ic_interrupt	0 = will not interrupt when completed 1 = will interrupt when completed
10	jump	0 = current instruction is not a jump instruction 1 = current instruction is a jump instruction
11	prefetched	0 = the next instruction is not prefetched into the prefetch buffer 1 = the next instruction is prefetched into the prefetch buffer

Table 1.14 ic_stat Register Fields

Field	Name	Description
12	stream	currently executing (non-overlapped instruction) stream: 0 = stream A 1 = stream B
13	pref_stream	stream of instruction prefetched: 0 = stream A 1 = stream B
14	condition	0 = jump condition not met 1 = jump met
16	a_wait	0 = A stream is running normally 1 = A stream paused because semaphore is taken
17	a_async	0 = No asynchronous transfer in progress in stream A 1 = Asynchronous transfer in progress in stream A
18	a_busy	0 = A stream not busy 1 = A stream busy
19	a_sem	A stream register semaphore status: 0 = semaphore not claimed by anyone or claimed by hardware 1 = semaphore held externally The agent receiving the semaphore will see a "0" in this bit.
20	a_lock	0 = A stream not locked in 1 = A stream locked in
21	a_primed	0 = A stream "sequence number completed" interrupt not primed 1 = A stream "sequence number completed" interrupt primed
22	a_paused	A stream pause 0 = A stream is not paused due to interrupt or error 1 = A stream is paused due to interrupt or error rethe coprocessor execution by writing 1
23	a_ol_primed	0 = A stream "overlapped instruction sequence number completed" interrupt not primed 1 = A stream "overlapped instruction sequence number completed" interrupt primed
24	b_wait	0 = B stream is running normally 1 = B stream paused because semaphore is taken
25	b_async	0 = No asynchronous transfer in progress in stream B 1 = Asynchronous transfer in progress in stream B
26	b_busy	0 = B stream not busy 1 = B stream busy

Table 1.14 ic_stat Register Fields

Field	Name	Description
27	b_sem	B stream register semaphore status: 0 = semaphore not claimed by anyone or held by the coprocessor 1 = semaphore held externally The agent receiving the semaphore will see a "0" in this bit.
28	b_lock	0 = B stream not locked in 1 = B stream locked in
29	b_primed	0 = B stream "sequence number completed" interrupt not primed 1 = B stream "sequence number completed" interrupt primed
30	b_paused	B stream pause 0 = B stream is not paused due to interrupt or error 1 = B stream is paused due to interrupt or error rethe coprocessor execution by writing 1
31	b_ol_primed	0 = B stream "overlapped instruction sequence number completed" interrupt not primed 1 = B stream "overlapped instruction sequence number completed" interrupt primed

n. ic_err_int

This register contains active high flags indicating whether an error or interrupt has occurred within the IC. Each bit can be cleared by writing a '1'.

The reset value of this register is 0x00000000.

Table 1.15 Ic_err_Int Register Fields

Field	Name	Description
8	a_ill_err	A stream illegal instruction error
16	b_ill_err	B stream illegal instruction error
24	a_snc_int	A stream "sequence number completed" interrupt
25	a_ic_int	A stream "instruction completed" interrupt
26	a_sns_int	A stream "sequence number the coprocessorted" interrupt
27	a_is_int	A stream "instruction the coprocessorted" interrupt
28	b_snc_int	B stream "sequence number completed" interrupt
29	b_ic_int	B stream "instruction completed" interrupt
30	b_sns_int	B stream " sequence number the coprocessorted" interrupt
31	b_is_int	B stream " instruction the coprocessorted" interrupt

o. **ic_err_int_en**

This register contains the masks used to enable the various errors or interrupts and has a reset value of 0x00000000.

Table 1.16 **ic_err_int_en** Register Fields

Field	Name	Description
8	a_ill_err_en	A stream illegal instruction error enable
16	b_ill_err_en	B stream illegal instruction error enable
24	a_snc_int_en	A stream "sequence number completed" interrupt enable
25	a_ic_int_en	A stream "instruction completed" interrupt enable
26	a_sns_int	A stream "sequence number the coprocessor sorted" interrupt enable
27	a_is_int	A stream "instruction the coprocessor sorted" interrupt enable
28	b_snc_int	B stream "sequence number completed" interrupt enable
29	b_ic_int	B stream "instruction completed" interrupt enable
30	b_sns_int	B stream "sequence number the coprocessor sorted" interrupt enable
31	b_is_int	B stream "instruction the coprocessor sorted" interrupt enable

p. **ic_ipa**

This register holds the most significant 30 bits of the next virtual address to be used to fetch an instruction for the A stream. The least two significant bits are assumed to be zero as instructions must be word aligned.

The reset value of this register is 0x00000000

Table 1.17 **ic_ipa** Register Fields

Field	Name	Description
[31:0]	ipa[31:2]	A stream instruction pointer

q. **ic_tda**

This register holds the "to do" value for the A stream. This is a 32 bit (wrapping) sequence number up to which valid instructions exist.

The reset value of this register is 0x00000000

Table 1.18 **ic_tda** Register Fields

Field	Name	Description
[31:0]	tda[31:0]	A stream "to do" value

r. **ic_fna**

This register holds the "finished" value for the A stream. This is a 32 bit (wrapping) sequence number indicating the last completed instruction.

The reset value of this register is 0x00000000.

Table 1.19 ic_fna Register Fields

Field	Name	Description
[31:0]	fna[31:0]	A stream "finished" value

s. ic_inta

This register holds the "interrupt" sequence number for the A stream. This is a 32 bit (wrapping) sequence number at which to "interrupt at" if the mechanism is enabled and primed.

The reset value of this register is 0x00000000

Table 1.20 ic_inta Register Fields

Field	Name	Description
[31:0]	inta[31:0]	A stream "interrupt" value

t. ic_loa

This register holds the 32 bit (wrapping) sequence number of the last overlapped instruction to be executed on the A stream.

The reset value of this register is 0x00000000

Table 1.21 ic_loa Register fields

Field	Name	Description
[31:0]	loa[31:0]	A stream "last overlapped instruction sequence" value

u. ic_ipb

This register holds the most significant 30 bits of the next virtual address to be used to fetch an instruction for the B stream. The least two significant bits are assumed to be zero as instructions must be word aligned.

The reset value of this register is 0x00000000

Table 1.22 ic_ipb Register Fields

Field	Name	Description
[31:0]	ipb[31:2]	B stream instruction pointer

v. ic_tdb

This register holds the "to do" value for the B stream. This is a 32 bit (wrapping) sequence number up to which valid instructions exist.

The reset value of this register is 0x00000000

Table 1.23 ic_tdb Register Fields

Field	Name	Description
[31:0]	tdb[31:0]	B stream "to do" value

w. ic_fnb

This register holds the "finished" value for the B stream. This is a 32 bit (wrapping) sequence number indicating the last completed instruction.

The reset value of this register is 0x00000000.

Table 1.24 ic_fnb Register Fields

Field	Name	Description
[31:0]	fnb[31:0]	B stream "finished" value

x. ic_intb

This register holds the "interrupt" sequence number for the B stream. This is a 32 bit (wrapping) sequence number at which to "interrupt at" if the mechanism is enabled and primed.

The reset value of this register is 0x00000000

Table 1.25 ic_intb Register Fields

Field	Name	Description
[31:0]	intb[31:0]	B stream "interrupt" value

y. ic_lob

This register holds the 32 bit (wrapping) sequence number of the last overlapped instruction to be executed on the B stream.

The reset value of this register is 0x00000000

Table 1.26 ic_lob Register fields

Field	Name	Description
[31:0]	lob[31:0]	B stream "last overlapped instruction sequence" value

z. Ic_sema

This register is an alias for the lc_stat register with the side effect that a read of this register is a request for the A stream register semaphore.

aa. ic_semb

This register is an alias for the lc_stat register with the side effect that a read of this register is a request for the B stream register semaphore.

Input Interface Switch Registers

ab. iis_cfg

Table 1.27 iis_cfg Register Fields

Field(bits)	Name	Definition
[31:14]		Reserved
[13:12]	po_p	priority of PO (0 = lowest, 2 = highest)
[11:10]	dcc_p	priority of DCC (0 = lowest, 2 = highest)
[9:8]	ic_p	priority of IC (0 = lowest, 2 = highest)
[7:4]		Reserved
[3]	S	0 = not in self test mode 1 = self test mode
[2]	B	0 = not bypassed 1 = bypassed
[1]	E	0 = disabled 1 = enabled

ac. iis_stat

Table 1.28 iis_stat Register Fields

Field(bits)	Name	Definition
[31:25]		Reserved
[24]	i_prefetch_po	state of i_prefetch_po signal
[23]	i_prefetch_dcc	state of i_prefetch_dcc signal
[22]	i_prefetch_ic	state of i_prefetch_ic signal
[21:20]	lmc_po_lpri	rotating priority of PO arbitrating for the LMC
[19:18]	lmc_dcc_lpri	rotating priority of DCC arbitrating for the LMC
[17:16]	lmc_ic_lpri	rotating priority of IC arbitrating for the LMC
[15:14]	eic_po_lpri	rotating priority of PO arbitrating for the EIC
[13:12]	eic_dcc_lpri	rotating priority of DCC arbitrating for the EIC
[11:10]	eic_ic_lpri	rotating priority of IC arbitrating for the EIC
9	lmc_gnt_po	1 = PO is granted control of LMC
8	lmc_gnt_dcc	1 = DCC is granted control of LMC
7	lmc_gnt_ic	1 = IC is granted control of LMC
6	eic_gnt_po	1 = PO is granted control of EIC
5	eic_gnt_dcc	1 = DCC is granted control of EIC
4	eic_gnt_ic	1 = IC is granted control of EIC
3	valid_req_po	1 = legal request made to IIS by PO
2	valid_req_dcc	1 = legal request made to IIS by DCC

Table 1.28 iis_stat Register Fields

Field(bits)	Name	Definition
1	valid_req_ic	1 = legal request made to IIS by IC
0	i_source	state of i_source signal

ad. iis_err_int

Table 1.29 iis_err_int Register Fields

Field(bits)	Name	Definition
[31:16]	Interrupt	interrupt condition
[15:0]	error	error condition : bit 3 = IBus request made to IIS while disabled bit 2 = read request from illegal the coprocessor location from Pixel Organiser bit 1 = read request from illegal the coprocessor location from Data Cache Controller bit 0 = read request from illegal the coprocessor location from Instruction Controller

ae. iis_err_int_en

Table 1.30 iis_err_int_en Register Fields

Field(bits)	Name	Definition
[31:16]	int_mask	interrupt mask
[15:0]	err_mask	error mask : bit 3 = IBus request made to IIS while disabled bit 2 = read request from illegal the coprocessor location from Pixel Organiser bit 1 = read request from illegal the coprocessor location from Data Cache Controller bit 0 = read request from illegal the coprocessor location from Instruction Controller

af. iis_ic_addr

Table 1.31 iis_ic_addr Register Fields

Field(bits)	Name	Definition
[31:0]	ic_addr	Address for data currently requested by IC

ag. iis_dcc_addr

Table 1.32 iis_dcc_addr Register Fields

Field(bits)	Name	Definition
[31:0]	dcc_addr	Address for data currently requested by DCC

ah. iis_po_addr

Table 1.33 iis_po_addr Register Fields

Field(bits)	Name	Definition
[31:0]	po_addr	Address for data currently requested by PO

ai. iis_burst

Table 1.34 iis_burst Register Fields

Field(bits)	Name	Definition
[13:0]		Reserved
[21:16]	po_burst	Burst length from PO
[13:8]	dcc_burst	Burst length from DCC
[5:0]	ic_burst	Burst length from IC

aj. iis_base_addr

Table 1.35 iis_base_addr Register Fields

Field(bits)	Name	Definition
[31:12]	the coprocessor_base [19:0]	Base address of the coprocessor memory object in memory map. the coprocessor memory object is aligned on a page boundary.
[11:0]		Reserved

ak. iis_test

Table 1.36 iis_test Register Fields

Field(bits)	Name	Definition
[31:4]		Reserved
[3:0]	test_mode[3:0]	0 = IC -> EIC, PO -> LMC 0 = IC -> EIC, DCC -> LMC 0 = PO -> EIC, IC -> LMC 0 = PO -> EIC, DCC -> LMC 0 = DCC -> EIC, IC -> LMC 0 = DCC -> EIC, PO -> LMC

External Interface Controller Registers

al. eic_cfg

Table 1.37 eic_cfg Register Fields

Field(bits)	Name	Definition
0	reserved	
1	enable	Enables all EIC operation as a master on PCI and Generic buses.
2	bypass	
3	reserved	Enables the operation of one or more of the EIC bypass modes.
5:4	pci_arb	PCI Bus Arbitration: 00 = Fair arbitration 01 = IBus has priority 10 = RBus has priority 11 = Fair arbitration
6	pci_ibus_min_len	
7	pci_rbus_min_len	
8	pci_mrl_mrm_en	Enables use of PCI memory read line and memory read multiple modes.
10:9	gen_arb	
11	gen_clk	0 = Generic bus clock is 1/4 of c1k 1 = Generic bus clock is 1/2 of c1k
21:12	mmu_mask	Mask bits that determine the page size used in the MMU.
29:24	hash_func	Determine the hash function used for page table lookups.

Table 1.37 eic_cfg Register Fields

Field(bits)	Name	Definition
30	hash_bypass	if 1 then bypass the hash function completely, so that the page table index is taken from the bottom 13 bits of the virtual page number (regardless of page size).
31	mmu_bypass	Forces the MMU to do null mapping on all requests.

an. eic_stat

Note: bits in this register can change at any time.

Table 1.38 eic_stat Register Fields

Field(bits)	Name	Definition
0	ibus_pci_act	Indicates an active request from the IBus to the PCI bus.
1	rbus_pci_act	Indicates an active request from the RBus to the PCI bus.
2	mmu_act	Indicates the MMU is in the middle of an operation.
3	gen_act	Indicates an active request to the Generic bus.
4	ibus_pci_gnt	IBus granted to PCI bus
5	rbus_pci_gnt	Rbus granted to PCI bus
6	mmu_pci_gnt	MMU service granted to PCI bus
31:4	Reserved	

an. eic_err_int

Error and interrupt bits of the eic_err_int register can only be set by the EIC, and can only be reset by software. Normal error and interrupt bits are reset by writing a 1 to that bit. Error bits which are copies of PCI configuration register bits must be cleared by writing to those registers - writing the eic_err_int copy of the bit has no effect.

Table 1.39 eic_err_int Register Fields

Field(bits)	Name	Definition
0	page_fault	Set when a page fault error occurs.
1	prot_fault	Set when a protection fault error occurs.
2	gen_err	Set when the Generic bus error pin is asserted for at least one gen_clk cycle.
3	gen_addr_err	Set when a Generic bus burst request crosses outside the Generic bus memory region.
4	rwd_err	Request when disabled - set when the EIC receives an RBus or IBus request when its enable bit is zero.
5	target_abort_err	A copy of the Received Target Abort bit in the PCI configuration space.

Table 1.39 eic_err_int Register Fields

Field(bits)	Name	Definition
6	master_abort_err	A copy of the Signalled Target Abort bit in the PCI configuration space.
7	reserved	
8	data_parity_exp	A copy of the Detected Data Parity Error bit in the PCI configuration space.
9	gen_exp	Set when the gen_exp_1 pin is asserted for at least one gen_clk cycle.
23:10	reserved	
24	gen_int	Set when the gen_int_1 pin is asserted for at least one gen_clk cycle.
31:25	Reserved	

ao. eic_err_int_en

Table 1.40 eic_err_int Register Fields

Field(bits)	Name	Definition
0	page_fault	Enables the corresponding error bit to cause the assertion of c_err.
1	prot_fault	Enables the corresponding error bit to cause the assertion of c_err.
2	gen_err	Enables the corresponding error bit to cause the assertion of c_err.
3	gcn_addr_err	Enables the corresponding error bit to cause the assertion of c_err.
4	rwd_err	Enables the corresponding error bit to cause the assertion of c_err.
5	target_abort_err	Enables the corresponding error bit to cause the assertion of c_err.
6	master_abort_err	Enables the corresponding error bit to cause the assertion of c_err.
7	reserved	
8	data_parity_exp	Enables the corresponding error bit to cause the assertion of c_exp.
9	gen_exp	Enables the corresponding error bit to cause the assertion of c_exp.
23:10	reserved	
24	gen_int	Enables the corresponding error bit to cause the assertion of c_int.
31:25	Reserved	

ap. eic_test

Table 1.41 eic_test Register Fields

Field(bits)	Name	Definition
31:0	Not yet defined	

aq. eic_pob

Table 1.42 eic_pob Register Fields

Field(bits)	Name	Definition
0	gen_pob0	Value of the generic bus programmable output pin pob0
1	gen_pob1	Value of the generic bus programmable output pin pob1

ar. eic_high_addr

Table 1.43 eic_dual_addr Register Fields

Field(bits)	Name	Definition
31:0	high_addr	Offset of 4GB region in host memory in which the coprocessor data structures exist (for dual address cycles).

as. eic_wtlb_v

Table 1.44 eic_wtlb_v Register Fields

Field(bits)	Name	Definition
1:0	pt_flags	The flags associated with the TLB entry to be written.
11:2	Reserved	
31:12	vpn	Virtual Page number to be written or invalidated.

at. eic_wtlb_p

Table 1.45 eic_wtlb_p Register Fields

Field(bits)	Name	Definition
0	inv_all	Causes the entire TLB to be invalidated. The EIC clears this bit after performing the TLB operation.
1	inv_entry	Invalidates a TLB entry with the virtual page number specified. The EIC clears this bit after performing the TLB operation.
2	write_lru	Writes a TLB entry to the LRU location with the specified virtual page number, physical page number and control bits. The EIC clears this bit after performing the TLB operation.
3	write	Write the TLB entry specified by the Entry Number field with the specified virtual page number, physical page number and flag bits. The EIC clears this bit after performing the TLB operation.
7:4	ent_num	Entry number to be written by Write Entry operation.
31:12	ppn	Physical Page number to be written.

au. eic_mmu_v

Note: values in this register can change at any time, unless the MMU is disabled by a page fault error or an MMU to PCI bus error.

Table 1.46 eic_mmu_v Register Fields

Field(bits)	Name	Definition
3:0	mmu_lru	Current LRU location
4	mmu_hit	mmu_hit is set when that last TLB access resulted in a hit without any page table access
5	mmu_retry_hit	v_mmu_retry_hit is set when that last TLB access resulted in a hit after a page table access
11:6	Reserved	
31:12	mmu_lvpn	The most recent virtual page number sent to the MMU for translation.

av. eic_mmu_p

Note: values in this register can change at any time, unless the MMU is disabled by a page fault error or an MMU to PCI bus error.

Table 1.47 eic_mmu_p Register Fields

Field(bits)	Name	Definition
31:0	mmu_lpta	The most recent page table physical address fetched by the MMU.

aw. eic_ip_addr

Note: values in this register can change at any time, unless the IBD is disabled by an IBus to PCI error.

Table 1.48 eic_ip_addr Register Fields

Field(bits)	Name	Definition
31:0	libpa	Physical address of the most recent IBus access to the PCI Bus.

ax. eic_rp_addr

Note: values in this register can change at any time, unless the RBR is disabled by an RBus to PCI error.

Table 1.49 eic_rp_addr Register Fields

Field(bits)	Name	Definition
31:0	lrbpa	Physical address of the most recent RBus access to the PCI Bus.

ay. eic_ig_addr

Note: values in this register can change at any time, unless the GBC is disabled by a Generic Bus error.

Table 1.50 eic_ig_addr Register Fields

Field(bits)	Name	Definition
31:0	libga	Address of the most recent IBus access to the Generic Bus.

az. eic_rg_addr

Note: values in this register can change at any time, unless the GBC is disabled by a Generic Bus error.

Table 1.51 eic_rg_addr Register Fields

Field(bits)	Name	Definition
31:0	lrbga	Address of the most recent RBus access to the Generic Bus.

PCIBus Configuration Space Alias

The PCIBus Configuration Space which consists of 16 words is aliased to the registers at addresses 0xC0 to 0xCF.

Local Memory Controller Registers

ba. Imi_cfg

Table 1.52 Imi_cfg Register Fields

Field(bits)	Name	Description
[31:24]	ref_interval	Refresh interval in multiples of 4-clk periods
[23:22]	ro_prio	RO access priority (3 = highest)
[21:20]	pic_prio	PIC access priority (3 = highest)
[19:18]	ic_prio	IC access priority (3 = highest)
[17:16]	iis_prio	IIS access priority (3 = highest)
[15:13]	rearb_interval	Re-arbitration interval in words (2^n words)
[12]	mem_enable	Memory enable (1 = memory installed)
[11]	banks	Number of DRAM banks connected (0=1, 1=2)
[10]	iis_flush	Flush the IIS prefetch data (self clearing)
[9]	iis_prefetch	IIS prefetching enable
[8]	refresh_en	Enable auto (CBR) refreshing of DRAM
[7]	seamless	Enable seamless SDRAM page faulting
[6]		reserved
[5:4]	bypass_mode	Bypass mode select: 00 = Rbus 01 = Abus write 10 = Abus read 11 = Ibus
[3]	S	Self test mode
[2]	B	Bypass enable
[1]	E	Enable module
[0]		Reserved

This register contains many configuration and control bits used to define LMC operation mode and parameters. Bits that refer specifically to SDRAM operation have no effect when the sdram_1 pin is high.

The register has a reset value of 0x20000100 setting the refresh interval to 3.2 us at a clk in frequency of 80 MHz. All special modes and features are disabled at power up and all access priorities are equal and set to zero. Refreshing is enabled at reset, but the rest of the module is disabled (E=0). Refreshing is not affected by the E bit.

bb. Imi_stat

Table 1.53 Imi_stat Register Fields

Field(bits)	Name	Description
[31]	ro_ca	RO module cycle active
[30]	pic_ca	PIC module cycle active
[29]		reserved
[28]	iis_ca	IIS module cycle active
[27]	ro_cp	RO module cycle pending
[26]	pic_cp	PIC module cycle pending
[25]		reserved
[24]	iis_cp	IIS module cycle pending
[23:22]		Reserved
[21:16]	stateA	Internal control state A
[15:13]		Reserved
[13:8]	stateB	Internal control state B
[7]	rd_active	Reading DRAM
[6]	wr_active	Writing DRAM
[5]	rf_active	Refresh in progress
[4]	rf_pending	Refresh pending
[3]	iis_pre_active	IIS prefetch active
[2:1]		Reserved
[0]	sdram	State of the sdram_1 input pin

The status register contains module active and pending bits as well as internal state machine state information. The state machine is clocked at twice the Cbus interface rate, so two fields are necessary to hold the state information, one for each of the last two 80 MHz clock cycles.

All bits within this register are read only.

bc. Imi_err_int

Table 1.54 Imi_err_int Register Fields

Field(bits)	Name	Description
[31:24]	interrupt	Interrupt status bits
[23:8]	exception	Exception status bits
[7:0]	error	Error status bits

The error and interrupt status register holds interrupt, exception and error condition information. The register is read / write where a read will return the status information and writing

one to a particular bit will cause that bit to be reset. Writing a zero to any bit has no effect on that bit.

Table 1.55 Interrupt Bit Definitions

Bit	Description
24	Refresh Interrupt. Occurs once per refresh interval (64 ms typ.)
25	Refresh not serviced prior to subsequent refresh request

Table 1.56 Error Bit Definitions

Bit	Description
0	Access to DRAM when no DRAM enabled
1	Access to DRAM while module disabled
2	Rbus Address Error
3	Abus Address Error
4	Cbus Address Error
5	Ibus Address Error

This register should have a reset value of 0x00000000 indicating that no interrupts or errors are asserted. Reserved bits are always zero and will never change state.

bd. lmi_err_int_en Register

Table 1.57 lmi_err_int_en Register Fields

Field(bits)	Name	Description
[31:24]	int_mask	Interrupt mask bits
[23:8]	exp_mask	Exception mask bits
[7:0]	err_mask	Error mask bits

The error, exception and interrupt enable register is used to selectively enable and disable error, exception and interrupt signals. The register is read / write. This register is used to enable on a bit-by-bit basis each of the error, exception and interrupt signals in the lmi_err_int register. There is a one-to-one correspondence between the bits in this register and the bits in the lmi_err_int register. If a particular bit in the lmi_err_int_en register is set high then the corresponding bit in the lmi_err_int register will be enabled, and if it is high, the LMC module error, exception or interrupt output signal, c_err, c_exp or c_int will be asserted. If a particular lmi_err_int_en bit is cleared to zero then the corresponding bit in the lmi_err_int register is disabled and cannot cause the assertion of the c_err, c_exp or c_int signal. Since there are no exceptions in the LMC, the exp_mask bits in this register have no effect and are all reserved.

This register has a reset value of 0x00000000 which disables all error and interrupt sources. Unused bits are always zero and cannot be set high.

be. Imi_dcfg

Table 1.58 Imi_dcfg Register Fields

Field(bits)	Name	Description
[31:29]	row_bits	Number of DRAM row address bits ($n + 6$)
[28:26]	column_bits	Number of DRAM column address bits ($n + 6$)
[25]	edo_dram	EDO type DRAM (0 = I-P, 1 = EDO)
[24:22]	precharge_enable	Precharge enable address bit ($n + 5$)
[21:19]	precharge_bank	Precharge bank address bit ($n + 5$)
[18:17]	cas_lo	Number of CAS low clock cycles (CAS pulse width) (1 to 4, 0 = 1)
[16]	cas_hi	Number of CAS high clock cycles (CAS pre-charge) (1 to 2, 0 = 1)
[15:13]	ras_lo	Minimum RAS low clock cycles (1 to 8, 0 = 1)
[12:11]	ras_hi	Minimum RAS high clock cycles (RAS pre-charge) (1 to 4, 0 = 1)
[10:9]	ras_cas	RAS to CAS delay in clock cycles (2 to 6, 0 = 2)
[8:7]	cas_lat	SDRAM CAS latency in clock cycles: 0, 1 = 1 clock 2 = 2 clocks 3 = 3 clocks
[6:0]		Reserved

This configuration register holds all the design parameters that determine the size and configuration of the DRAM chips used.

This register has a reset value of 0x0007FF80 which sets all timing constraints to their maximum values.

bf. Imi_mode Register

Table 1.59 Imi_mode Register Fields

Field(bits)	Name	Description
[31:14]		Reserved
[13]	initialize	Initialize SDRAMs (program mode register, self clearing)
[12:0]	sdram_mode	SDRAM mode bits (written to SDRAM mode register over the address pins by initialize SDRAM command)

This configuration register holds the information that is written into the SDRAM mode register as part of the initialization procedure. This register may be read or written at any time and may be written to the SDRAM by setting the initialize bit.

This register has a reset value of 0x0037. This is a useful default value that is required immediately after precharging at power up or after a level 1 reset. This sets read latency to 3 clocks and burst length to full page with sequential wrap.

After any reset, the initialize bit will be set if the `sdram_1` pin is low, to initially program the SDRAM mode register. After the mode register write is performed, this bit will be cleared to zero automatically.

Peripheral Interface Controller Registers

bg. `pic_cfg`

Table 1.60 `pic_cfg` Register Fields

Field (bits)	Name	Definition
0	reserved	
1	enable	Enables all PIC operation.
2	bypass	
3	reserved	
4	big_endian	Causes all byte packing/unpacking to use big-endian order, i.e. bits 31:24 first, bits 7:0 last.
5	video mode	0 = Video Output 1 = Video Input
7:6	reserved	
8	gate_out_clock	Used in Video Output mode. Causes the output clock to be gated with <code>vclk_en_in_1</code> .
9	inv_out_clock	Used in Video Output mode. Causes the output clock to be inverted with respect to <code>vclk_in</code> .
10	use_default_data	Used in Video Output mode. If this bit is set, the value in the <code>default_data</code> register bit is driven onto the output data pins for cycles when data is not valid. Otherwise, the next valid data byte is driven if available.
11	default_data	Used in Video Output mode. The default data used if <code>use_default_data</code> is enabled.
12	data_en_source	Used in Video Output mode. When 1, <code>vdata_en_out_1</code> is asserted when <code>vdata_en_in_1</code> is sampled asserted. Otherwise, <code>vdata_en_out_1</code> is asserted when <code>vclk_en_in_1</code> is sampled asserted.
13	del_data	Used in Video Output mode. Causes the output data to be delayed one cycle with respect to <code>vdata_en_out_1</code> .
14	gate_with_rdy	Used in Video Output mode. Causes the output clock to be gated with <code>vr dy_1</code> .
15	reserved	
16	forward_act_dis	Disables the assertion of <code>c_active</code> due to data in the output FIFO.
17	reverse_act_dis	Disables the assertion of <code>c_active</code> due to data in the input FIFO.
18	forward_priority	Relative priority of forward transfers when in a bidirectional mode.

Table 1.60 pic_cfg Register Fields

Field(bits)	Name	Definition
19	reverse_priority	Relative priority of reverse transfers when in a bidirectional mode.
20	forward_invert_data	Invert output data signals
21	reverse_invert_data	Invert input data signals
22	forward_invert_control	Invert output control signals
23	reverse_invert_control	Invert output control signals
30:24	cbus_timer	Maximum time for which a CBus cycle to the input or output data stream can be stalled before signalling an error. Specified in multiples of 8 clk cycles.
31	cbus_timer_disable	Disables the operation of the CBus timer.

bh. pic_stat

Table 1.61 pic_stat Register Fields

Field(bits)	Name	Definition
0	abus_act	Set when an ABus transaction is pending or in progress.
1	rbus_act	Set when an RBus transaction is pending or in progress.
2	reverse_act	Set a when a reverse data transfer is active.
3	cent_cfg	Copy of the external cent_cfg pin.
4	forward_full	Set when the PIC's output FIFO is full.
5	forward_subword	Set when the PIC's output FIFO has space for more than zero but less than four bytes.
6	forward_empty	Set when the PIC's output FIFO is empty.
7	reserved	
8	reverse_full	Set when the PIC's input FIFO is full.
9	reverse_empty	Set when the PIC's input FIFO is empty.
10	reverse_subword	Set when the PIC's input FIFO contains more than zero but less than four bytes.
15:11	reserved	
16	pic_plh	The state of the pic_plh pin.
17	pic_ack_1	The state of the pic_ack_1 pin.
18	pic_busy	The state of the pic_busy pin.
19	pic_perror	The state of the pic_perror pin.
20	pic_select	The state of the pic_select pin.
21	pic_fault_1	The state of the pic_fault_1 pin.
31:24	pic_data_in	The state of the pic_data_in signal.

bi. pic_err_int

Error and interrupt bits of the pic_err_int register can only be set by the PIC, and can only be reset by software. Each bit is reset by writing a 1 to that bit.

Table 1.62 pic_err_int Register Fields

Field(bits)	Name	Definition
0	forward_err	Forward transfer error. Set if the PIC receives output data to transfer when configured for an input-only mode.
1	rwd_err	Request when disabled error. Set if the PIC is disabled when it receives request for an RBus transaction, or any register-initiated operation (ABus transfer, Centronics command, etc.).
2	timeout_err	Peripheral timeout error.
8	reverse_exp	Reverse transfer error. Set if software attempts to the coprocessor a reverse transfer when the PIC is in an output only mode.
9	cbus_exp	Set when a CBus cycle to the input or output data stream times out. When this bit is set, further CBus cycles to the input or output data streams are treated as reserved.
24	rev_comp_int	Set when the byte-count limit has been reached in a reverse transfer.
25	rev_data_int	Set when data is available from a reverse transfer and there is no active bus transaction to indicate its destination.
26	abus_comp_int	Set when abus transfer completes
27	timer_int	Set when timer 0 or 1 expires
28	comp_fault_int	Set in Centronics compatible mode when the pic_fault_1 signal is asserted.
29	comp_perror_int	Set in Centronics compatible mode when the pic_perror signal is asserted.
30	ecp_rev_req_int	Set in Centronics ECP mode when the peripheral indicates that it has reverse data available.
31	no_peripheral_int	No peripheral interrupt

bj. pic_err_int_en

Table 1.63 pic_err_int_en Register Fields

Field(bits)	Name	Definition
0	forward_err	Enables the corresponding interrupt bit to cause the assertion of c_int.
1	rwd_err	Enables the corresponding interrupt bit to cause the assertion of c_int.
2	timeout_err	Enables the corresponding interrupt bit to cause the assertion of c_int.

Table 1.63 pic_err_int_en Register Fields

Field(bits)	Name	Definition
8	reverse_exp	Enables the corresponding interrupt bit to cause the assertion of c_int.
9	cbus_exp	Enables the corresponding interrupt bit to cause the assertion of c_int.
24	rev_comp_int	Enables the corresponding interrupt bit to cause the assertion of c_int.
25	rev_data_int	Enables the corresponding interrupt bit to cause the assertion of c_int.
26	abus_comp_int	Enables the corresponding interrupt bit to cause the assertion of c_int.
27	timer_int	Enables the corresponding interrupt bit to cause the assertion of c_int.
28	comp_fault_int	Enables the corresponding interrupt bit to cause the assertion of c_int.
29	comp_perror_int	Enables the corresponding interrupt bit to cause the assertion of c_int.
30	ecp_rev_req_int	Enables the corresponding interrupt bit to cause the assertion of c_int.
31	no_peripheral_int	Enables the corresponding interrupt bit to cause the assertion of c_int.

bk. pic_abus_cfg

Table 1.64 pic_abus_cfg Register Fields

Field(bits)	Name	Definition
23:0	ab_count	Number of bytes remaining to transfer to/from the LMC. Set initially by software, and decremented by the PIC as each byte is transferred.
27:24	ab_byte_en	Byte lanes to be used for ABus transfer
28	ab_type	1 = Transfer is a read (from peripheral port) 0 = Transfer is a write (to peripheral port)
29	ab_the coprocessor	the coprocessors the programmed ABus transfer. This bit is cleared by the PIC when the ABus transfer has been completed. If cleared by software, the ABus transfer is terminated.
31:30	reserved	

bl. pic_abus_addr

Table 1.65 pic_abus_addr Register Fields

Field(bits)	Name	Definition
31:0	ab_addr	The byte address at which the next ABus transaction should be the coprocessor.

bm. pic_cent_cfg

The pic_cent_cfg register contains read/write control signals and read-only status signal control all aspects of the interface when Centronics mode is enabled.

Table 1.66 pic_cent_cfg Register Fields

Field(bits)	Name	Definition
2:0	cen_cmd	Centronics Command: 000 = Return to Compatible 001 = Request Nibble 010 = Transfer Device ID using Nibble 011-100 = no-op 101 = Request ECP 110 = Transfer Device ID using ECP 111 = Abort
3	cen_exc	the coprocessors execute of programmed Centronics command. This bit is reset when the Centronics command completes.
4	cen_sis	This read-only bit shows the completion status of the previous Centronics command. A 1 indicates that the command was successful, a 0 that it was unsuccessful.
7:5	cen_rmode	Read-only bits showing current Centronics mode and operation based on the last negotiation: 000 = Compatible 001 = Compatible with Nibble enabled 010 = Transferring Device ID using Nibble 011 = Nibble Mode 100 = reserved 101 = ECP forward mode 110 = Transferring Device ID using ECP 111 = ECP reverse mode
8	cen_direct	Bypasses the Centronics control logic and allows the software to directly control the output bits by writing and reading this register.
9	cen_host_rec_en	Enables host recovery mechanism in ECP mode.
15:10	reserved	
23:16	cen_tim	Timer value, in ≤ 1 k cycles, for all Centronics setup, hold and pulse times.
31:24	cen_per_tim	Timer value, in lots of 64k ≤ 1 k cycles, when waiting for a Centronics peripheral response.

bn. pic_cent_dir

Table 1.67 pic_cent_dir Register Fields

Field(bits)	Name	Definition
0	pic_data_oe	Direct control bit for the pic_data_oe signal.
1	pic_strobe_l	Direct control bit for the pic_strobe_l signal.
2	pic_autofd_l	Direct control bit for the pic_autofd_l signal.
3	pic_init_l	Direct control bit for the pic_init_l signal.
4	pic_selectin_l	Direct control bit for the pic_selectin_l signal.
5	pic_buf_en_l	Direct control bit for the pic_buf_en_l signal.
6	pic_buf_dir	Direct control bit for the pic_buf_dir signal.
7	reserved	
15:8	pic_data_out	Direct control bit for the pic_data_out signal.
31:16	reserved	

bo. pic_reverse_cfg

Table 1.68 pic_reverse_cfg Register Fields

Field(bits)	Name	Definition
23:0	rev_count	Number of bytes remaining to transfer from the external peripheral. Set initially by software, and decremented by the PIC as each byte is transferred.
24	rev_use_count	Causes the reverse transfer to be terminated after the specified number of bytes have been transferred.
25	rev_the coprocessor	the coprocessor sorts the programmed reverse transfer. If the rev_use_count bit is set, this bit is cleared by the PIC when the reverse transfer has been completed. The software can clear this bit to stop the reverse transfer at any time.
31:26	reserved	

bp. pic_timer0

Table 1.69 pic_timer0 Register Fields

Field(bits)	Name	Definition
31:0	timer0	Timer value for first data timeout, in units of one clk.

bq. pic_timer1

Table 1.70 pic_timer1 Register Fields

Field(bits)	Name	Definition
31:0	timer1	Timer value for data-to-data timeout, in units of one clk.

Data Cache Controller Registers

br. dcc_cfg1

Table 1.71 dcc_cfg1 Register Fields

Field(bits)	Name	Definition
1	E	0=disabled 1=enabled
2	B	0=not bypassed 1=bypassed

bs. dcc_cfg2

Table 1.72 dcc_cfg2 Register Fields

Field(bits)	Name	Definition
0	Autoinv	0=not in auto-invalidate mode (value upon reset) 1=auto-invalidate cache at the end of each instruction
1	Autofill	0=not in auto-fill mode (value upon reset) 1=auto-fill cache the coprocessor at the address specified in dcc_addr. Cache is validated at the end of the operation. This bit is self-clearing.
2	Lock	0=data is written to cache RAM after data fetch (value upon reset) 1=data is not written to cache RAM after data fetch
3	always_hit	0=Tag memory and valid bits are checked (value upon reset) 1=Data is assumed to be always valid and resident in cache
4	always_miss	0=Tag memory and valid bits are checked (value upon reset) 1=Data is assumed to be not resident in cache

Table 1.72 dcc_cfg2 Register Fields

Field(bits)	Name	Definition
[10:8]	mode	current mode of caching operation: 0=random access- normal mode (value upon reset) 1=JPEG Encoding 2=slow JPEG Decoding 3=single channel colour space conversion 4=multi channel colour space conversion 5= matrix multiplication 6=64 bit mode 7=Invalidate all entries in cache. The DCC will revert to mode 1 when all the "line valid" bits have been cleared. This field is reset at the end of each instruction.
11	cache_miss_inst	current mode of caching operation: 0=Tag memory and valid bits are checked (value upon reset) 1=Data is assumed to be not resident in cache This bit is reset at the end of each instruction.

bt. dcc_stat

Table 1.73 dcc_stat Register Fields

Field(bits)	Name	Definition
[3:0]	error[3:0]	error condition: bit 0=attempted R/W operation to cache or Tag memory with the module disabled.
[p:q]	state[7:0]	internal state condition. Details to be determined

bu. dcc_err_int

Table 1.74 dcc_err_int Register Fields

Field(bits)	Name	Definition
0	error	attempted R/W operation to cache or Tag memory with the module disabled.

bv. dcc_err_int_en

Table 1.75 dcc_err_int_en Register Fields

Field(bits)	Name	Definition
0	error_en	enable corresponding error

bw. dcc_lv0

Table 1.76 dcc_lv0 Register Fields

Field(bits)	Name	Definition
[31:0]	lv0[31:0]	valid bits for lines 31 down to 0

bx. dcc_lv1

Table 1.77 dcc_lv1 Register Fields

Field(bits)	Name	Definition
[31:0]	lv1[31:0]	valid bits for lines 63 down to 32

by. dcc_lv2

Table 1.78 dcc_lv2 Register Fields

Field(bits)	Name	Definition
[31:0]	lv2[31:0]	valid bits for lines 95 down to 64

bz. dcc_lv3

Table 1.79 dcc_lv3 Register Fields

Field(bits)	Name	Definition
[31:0]	lv3[31:0]	valid bits for lines 127 down to 96

ca. dcc_addr

Table 1.80 dcc_addr Register Fields

Field(bits)	Name	Definition
[31:0]	dcc_addr	This is the base address register used for the special addressing modes. Note that different bits of this register are used depending on the operational mode.

cb. dcc_raddrb

Table 1.81 dcc_raddrb Register Fields

Field(bits)	Name	Definition
[31:0]	dcc_raddrb	This is the status of Operand Organiser B's address

cc. dcc_raddrc

Table 1.82 dcc_raddrc Register Fields

Field(bits)	Name	Definition
[31:0]	dcc_raddrc	This is the status of Operand Organiser C's address

cd. dcc_test

Table 1.83 dcc_test Register Fields

Field(bits)	Name	Definition
0	drive_odata	This bit is only valid when the test_mode bits are set to 0. 0=do not drive Operand Organiser data busses (value upon reset) 1=drive Operand Organiser data busses
1	o_ackb	This bit is only valid when the test_mode bits are set to 0. This bit is only useable when the DCC is in its test mode and the drive_odata bit is set. This bit is self-clearing. 0=do not assert o_ackb (value upon reset) 1=assert o_ackb for one clk cycle

Table 1.83 dcc_test Register Fields

Field(bits)	Name	Definition
2	o_ackc	This bit is only valid when the test_mode bits are set to 0. This bit is only useable when the DCC is in its test mode and the drive_odata bit is set. This bit is self-clearing. 0=do not assert o_ackc (value upon reset) 1=assert o_ackc for one clk cycle
[5:3]	test_mode	0=drive odata buses mode(value upon reset) 1=drive OOB address out to IIS, i_oe=0 2=drive OOC address out to IIS, i_oe=0 3=drive IIS data to o_dataab[31:0], i_oe=1 4=drive IIS data to o_dataac[31:0], i_oe=1

Operand Organiser Registers

Note that there are two identical Operand Organisers: Operand Organiser B and Operand Organiser C. The registers for these two Operand Organisers (which are identical) are described here.

ce. oon_cfg (oob_cfg=0x70, ooc_cfg=0x80)

Table 1.84 oon_cfg Register Fields

Field	Name	Description
[31:9]		Reserved
[8]	operate	the coprocessor bit for OO (self-cleared at end of each instruction) 0 = idle 1 = operate
[7:2]		Reserved
[1]	E	0 = disabled 1 = enabled
[0]		Reserved

cf. `oon_stat` (`oob_cfg=0x71`, `ooc_cfg=0x81`)

Table 1.85 `oon_stat` Register Fields

Field	Name	Description
[31:24]		Reserved
[23:16]	state[7:0]	internal state condition : bit 0 = OO active bit 1 = OO stalled (sequential mode only) bit 2 = FIFO empty bit 3 = FIFO full
[15:0]		Reserved

cg. `oon_err_int` (`oob_err_int=0x72`, `ooc_err_int=0x82`)

Table 1.86 `oon_err_int` Register Fields

Field	Name	Description
[15:0]	error	error condition : bit 0 = OBus request received by OO while disabled bit 1 = MDP and JPEG Coder requesting data simultaneously

ch. `oon_err_int_en` (`oob_err_int_en=0x73`, `ooc_err_int_en=0x83`)

Table 1.87 `oon_err_int_en` Register Fields

Field	Name	Description
[15:0]	error	error condition : bit 0 = OBus request received by OO while disabled bit 1 = MDP and JPEG Coder requesting data simultaneously

ci. oon_dmr (oob_dmr=0x74, ooc_dmr=0x84)

Table 1.88 oon_dmr Register Fields

Field	Name	Description
[31:30]	ls3	lane swap for byte 3 : 0 = from byte 0 1 = from byte 1 2 = from byte 2 3 = no swap
[29:28]	ls2	lane swap for byte 2 : 0 = from byte 0 1 = from byte 1 2 = no swap 3 = from byte 3
[27:26]	ls1	lane swap for byte 1 : 0 = from byte 0 1 = no swap 2 = from byte 2 3 = from byte 3
[25:24]	ls0	lane swap for byte 0 : 0 = no swap 1 = from byte 1 2 = from byte 2 3 = from byte 3
[23:20]	suben[3:0]	Substitution enables : 0 = do not perform substitution operation on this byte 1 = use value stored in oon_subst for this byte
[19:15]	replicate	Replication Count : Indicates number of <i>additional</i> data items to generate
[14:12]	normalise	Normalisation factor : 0 = 1 bit per channel 1 = 2 bits per channel 2 = 4 bits per channel 3 = 8 bits per channel 4 to 7 = 16 bits per channel
[11]		Reserved
[10:8]	bo[2:0]	Bit offset within byte of bit object
[7]	P	External Format : 0 = input objects are unpacked bytes 1 = input objects are a packed stream
[6:5]	if	Internal Format : 0 = pixels 1 = unpacked bytes 2 = packed bytes 3 = other

Table 1.88 oon_dmr Register Fields

Field	Name	Description
[4:3]	cc	Input Channel Count : 0 = 4 active channels 1 = 1 active channels 2 = 2 active channels 3 = 3 active channels
[2]	L	0 = immediate data in oo_said register 1 = direct addressing
[1:0]	what	Addressing Mode : 0 = bypass 1 = sequential 2 = tiling 3 = constant data

cj. oon_subst (oob_subst=0x75, ooc_subst=0x85)

Table 1.89 oon_subst Register Fields

Field	Name	Description
[31:0]	subst	value of substitution data

ck. oon_cdp (oob_cdp=0x76, ooc_cdp=0x86)

Table 1.90 oon_cdp Register Fields

Field	Name	Description
[31:0]	cdp	current data pointer

cl. oon_len (oob_len=0x77, ooc_len=0x87)

Table 1.91 oon_len Register Fields

Field	Name	Description
[31:0]	len	length of input operand stream

cm. oon_said (oob_said=0x78, ooc_said=0x88)

Table 1.92 oon_said Register Fields

Field	Name	Description
[31:0]	said	the coprocessor address or immediate data

cn. oon_tile (oob_tile=0x79, ooc_tile=0x89)

Table 1.93 oon_tile Register Fields

Field	Name	Description
[31:16]	offset[15:0]	offset into operand
[15:0]	tile_len[15:0]	length - 1, in bytes, of operand for tiling operation

Pixel Organiser Registers

co. po_cfg

Table 1.94 po_cfg Register Fields

Field	Name	Description
[26:24]	burst[2:0]	log base 2 of maximum burst length for input data. Data bursts are aligned to addresses of this value.
[22:21]	chan	Number of input channels for JPEG compression : 0, 1 = Single channel 2 = 3 channels 3 = 4 channels
[20]	SS	0 = no subsampling 1 = perform subsampling
[19]	F	Filter option for subsampling : 0 = no filtering (use every second byte) 1 = apply filtering (average)
[18:16]		Reserved
[15]	dst	destination for PBus data : 0 = MDP 1 = JPEG Coder
[14:12]	mode[2:0]	Operating mode for PO : 0 = idle 1 = sequential mode 2 = JPEG Compression 3 = CSC 4 to 7 = Convolution/Image Transformation
[11:4]		Reserved
[3]	S	0 = not in self test mode 1 = self test mode
[2]	B	0 = not bypassed 1 = bypassed
[1]	E	0 = disabled 1 = enabled
[0]		Reserved

cp. po_stat

Table 1.95 po_stat Register Fields

Field	Name	Description
[31:27]		Reserved
[26]	mu_v_miss	0 = no MUV miss 1 = MUV miss
[25]	po_stall	0 = PO not stalled 1 = current PO operation stalled
[24]	po_active	0 = PO not active 1 = PO active
[20]	sm_the copro- cessor	PO the coprocessor state
[19]	ack_rcvd	ibus ack received
[18]	ibus_req_made	ibus req made
[17]	fifo_full	FIFO full
[16]	fifo_empty	FIFO empty
[15:0]		Reserved

cq. po_err_int

Table 1.96 po_err_int Register Fields

Field	Name	Description
[7:0]	error	error condition : bit 0 = IBus i_ack asserted to PO while disabled bit 1 = both PO and RO using MUV RAM simultane- ously bit 2 = jpeg mode data overflow

cr. po_err_int_en

Table 1.97 po_err_int_en Register Fields

Field	Name	Description
[7:0]	err_mask	error mask : bit 0 = IBus i_ack asserted to PO while disabled bit 1 = both PO and RO using MUV RAM simultane- ously bit 2 = jpeg mode data overflow

cs. po_dmr

Table 1.98 po_dmr Register Fields

Field	Name	Description
[31:30]	ls3	lane swap for byte 3 : 0 = from byte 0 1 = from byte 1 2 = from byte 2 3 = no swap
[29:28]	ls2	lane swap for byte 2 : 0 = from byte 0 1 = from byte 1 2 = no swap 3 = from byte 3
[27:26]	ls1	lane swap for byte 1 : 0 = from byte 0 1 = no swap 2 = from byte 2 3 = from byte 3
[25:24]	ls0	lane swap for byte 0 : 0 = no swap 1 = from byte 1 2 = from byte 2 3 = from byte 3
[23:20]	suben[3:0]	Byte Substitution Enables : 0 = do not substitute data from po_subst register for this byte 1 = use po_subst register value for this byte
[19:15]	replicate[4:0]	Replication Count : indicates number of <i>additional</i> internal data items to generate
[14:12]	norm_factor[2:0]	Normalisation factor for input data : 0 = 1 bit per component 1 = 2 bits per component 2 = 4 bits per component 3 = 8 bits per component 4 to 7 = 16 bits per component
[11]		Reserved
[10:8]	bo[2:0]	bit offset within byte; Bit addressing is big endian
[7]	P	External format : 0 = input data consists of unpacked bytes 1 = input consists of a packed stream

Table 1.98 po_dmr Register Fields

Field	Name	Description
[6:5]	if[1:0]	Internal format of data passed to output : 0 = pixels 1 = unpacked bytes 2 = packed bytes 3 = other
[4:3]	cc[1:0]	Channel Count for input stream : 0 = 4 active channels 1 = 1 active channel 2 = 2 active channels 3 = 3 active channels
[2]	L	0 = immediate operand ("short" format) 1 = direct addressing ("long" format)
[1:0]	what[1:0]	DMU addressing mode : 0 = bypass 1 = sequential 2 = tiling 3 = constant

ct. po_subst

Table 1.99 po_subst Register Fields

Field	Name	Description
31:0	subst[31:0]	value of substitution data

cu. po_cdp

Table 1.100 po_cdp Register Fields

Field	Name	Description
31:0	cdp[31:0]	address of current data item

cv. po_len

Table 1.101 po_len Register Fields

Field	Name	Description
31:0	len[31:0]	length of the operand

cw. po_said

Table 1.102 po_said Register Fields

Field	Name	Description
[31:0]	ad	the coprocessor address of operand data (or immediate data value)

cx. po_idr

Table 1.103 po_idr Register Fields

Field	Name	Description
[31:0]	width[31:16] height[15:0]	width - 1, in pixels, of current image height - 1, in lines, of current image

cy. po_muv_valid

Table 1.104 po_muv_valid Register Fields

Field	Name	Description
[31:0]	valid	Valid bits for MUV lines

cz. po_muv

Table 1.105 po_muv Register Fields

Field	Name	Description
[31:11]	mu_v_base_addr	base address of MUV RAM

Main Data Path Registers

da. mdp_cfg

All bits are reset to 0.

Table 1.106 mdp_cfg Register Fields

Field	Name	Description
1	E	0 = disabled 1 = enabled
2	B	0 = not bypassed 1 = bypassed
3	T	0 = not in self test mode 1 = self test mode

Table 1.106 mdp_cfg Register Fields

Field	Name	Description
14	decode	0 = Instruction decoding disabled 1 = Instruction decoding enabled: mdp_op1 and mdp_op2 set up from mdp_cfg (self clearing)
15	operate	the coprocessor bit (self clearing)
17	word_input	0 = horizontal interpolation with byte inputs 1 = horizontal interpolation with word input
18	long_krnl	format of kernel descriptor 0 = short format 1 = long format
19	blend_gen	0 = blend generation not required 1 = blend generation required
31:20	Instruction[15:0]	Instruction major and minor opcodes

db. mdp_stat

All bits are reset to zero

Table 1.107 mdp_stat Register Fields

Field	Name	Description
0	po_valid	status of 'po_valid' signal
1	po_final	status of 'po_final' signal
2	po_stall	status of 'po_stall' signal
3	oob_valid	status of 'oob_ack' signal
4	oob_req	status of 'oob_req' signal
5	oob_pending	0 = no outstanding requests in OOB interface 1 = outstanding requests in OOB interface
6	ooc_valid	status of 'ooc_ack' signal
7	ooc_req	status of 'ooc_req' signal
8	ooc_pending	0 = no outstanding requests in OOC interface 1 = outstanding requests in OOC interface
9	ro_valid	status of 'ro_valid' signal
10	ro_final	status of 'ro_final' signal
11	ro_stall	status of 'ro_stall' signal
12:13	matmul_state[1:0]	matrix multiplication state 0 = idle 1 = multiplying the left half of matrix 2 = multiplying the right half of matrix
14:15	int_stat[1:0]	interpolation by fixed proportion states: 0 = cycle 0 1 = cycle 1 2 = cycle 2 3 = cycle 3

Table 1.107 mdp_stat Register Fields

Field	Name	Description
16	jump	ramp generator mode (in blend generation) 0 = step mode 1 = jump mode
17:18	addgen_state[1:0]	state of the Address Generation state machine 0 = idle 1 = fetching operand from PO 2 = generating address
19:20	ramp_state	state of ramp generation state machine 0 = idle 1 = determine mode of operation 2 = generating blend

dc. mdp_err_int

All bits are reset to zero.

Table 1.108 mdp_err_int Register Fields

Field	Name	Description
7:0	error[7:0]	error[0] = receive data from PO when disabled or MDP is not the coprocessor error[1] = receive data from OOB when disabled or MDP is not the coprocessor error[2] = receive data from OOC when disabled or MDP is not the coprocessor
24:8	exception[15:0]	exception[0] = underflow or overflow in clamping in channel 0 exception[1] = underflow or overflow in clamping in channel 1 exception[2] = underflow or overflow in clamping in channel 2 exception[3] = underflow or overflow in clamping in channel 3 exception[4] = x co-ordinate underflow (image transformations and convolutions) exception[5] = y co-ordinate underflow (image transformations and convolutions)

241

30?

dd. mdp_err_int_en

All bits are reset to zero.

Table 1.109 mdp_err_int_en Register Fields

Field	Name	Description
7:0	err_mask[7:0]	mask error condition 0 = masked 1 = not masked err_mask[0] = receive data from PO when disabled or MDP is not the coprocessor err_mask[1] = receive data from OOB when disabled or MDP is not the coprocessor err_mask[2] = receive data from OOC when disabled or MDP is not the coprocessor
24:8	exp_mask[15:0]	mask exception 0 = masked 1 = not masked exp_mask[0] = underflow or overflow in clamping in channel 0 exp_mask[1] = underflow or overflow in clamping in channel 1 exp_mask[2] = underflow or overflow in clamping in channel 2 exp_mask[3] = underflow or overflow in clamping in channel 3 exp_mask[4] = x co-ordinate underflow (image transformations and convolutions) exp_mask[5] = y co-ordinate underflow (image transformations and convolutions)

de. mdp_test

All bits are reset to 0

Table 1.110 mdp_test Register Fields

Field	Name	Description
3:0	test_data[3:0]	The source of data to RO under test mode 0-3 = RO data is from PO data (setting to 0, 1, 2 or 3 would have the same effect) 4 = RO data is from oob_data[31:0] 5 = RO data is from oob_data[63:32] 6 = RO data is from oob_data[95:64] 7 = RO data is from oob_data[127:96] 8 = RO data is from ooc_data[31:0] 9 = RO data is from ooc_data[63:32] 10 = RO data is from ooc_data[95:64] 11 = RO data is from ooc_data[127:96] 12-15 = reserved
25:4	reserved	

Table 1.110 mdp_test Register Fields

Field	Name	Description
29:26	output_delay[3:0]	the number of clock cycles between input and output - 2
31:30	skeletal_mode[1:0]	mode of skeletal MDP 0 = PO to RO with some delay 1 = PO -> OOB -> OOC -> RO with some delay 2 = the least significant byte of the data from PO, OOB and OOC are combined to give the result word to RO. 3 = reserved

df. mdp_op1

All bits are reset to zero

Table 1.111 mdp_op1 Register Fields

Field	Name	Description
2:0	ppb_modeA[2:0]	mode of multi-function block A in Preprocessing Block
5:3	ppb_modeB[2:0]	mode of multi-function block B in Preprocessing Block
8:6	pba_modeA[2:0]	mode of multi-function block A in Stage A Processing Block
11:9	pba_modeB[2:0]	mode of multi-function block B in Stage A Processing Block
14:12	pba_modeC[2:0]	mode of multi-function block C in Stage A Processing Block
17:15	pbb_modeA[2:0]	mode of multi-function block A in Stage B Processing Block
20:18	pbb_modeB[2:0]	mode of multi-function block B in Stage B Processing Block
23:21	pbb_modeC[2:0]	mode of multi-function block C in Stage B Processing Block

Table 1.111 mdp_op1 Register Fields

Field	Name	Description
27:24	inst_type[3:0]	Type of instruction: 0 = no_op (which implies that the MDP does not for that instruction) 1 = GCSC 2 = reserved 3 = Matrix Multiplication 4 = Convolutions 5 = Image Transformation 6 = reserved 7 = Half Toning 8 = Residual merging 9 = bypass (which implies that MDP only needs to pass data from PO to RO) 10 = Horizontal interpolation 11 = Vertical Interpolation 12-13 = compositing 14-15 = reserved
28	it_int	Interpolation required in Image transformation 0 = no interpolation 1 = interpolation
29	it_acc	Accumulation required in Image transformation 0 = no accumulation 1 = accumulation
30	comp_unpre	Un-pre-multiplication required in Compositing 0 = no un-pre-multiplication 1 = un-pre-multiplication
31	comp_blend	Blend generation required in compositing 0 = no blend generation 1 = blend generation

dg. mdp_op2

All bits are reset to zero.

Table 1.112 mdp_op2 Register Fields

Field	Name	Description
1:0	mul_A[1:0]	multiplicand of Operand A pixel 0 = 0 1 = 1 2 = opacityB 3 = -opacityB
3:2	mul_B[1:0]	multiplicand of Operand B pixel 0 = 0 1 = 1 2 = opacityA 3 = -opacityA

Table 1.112 mdp_op2 Register Fields

Field	Name	Description
4	reverse	0 = do not reverse operand in compositing 1 = reverse operand in compositing
5	addgen_mode	Address generation mode 0 = image transformation mode 1 = convolution mode
6	self_cfg	0 = long kernel descriptor 1 = short kernel descriptor, other parameters are self configured
7	reserved	
8	ag_the_coprocessor	address generation the coprocessor bit. It is cleared when the address generation is finished
9	bg_the_coprocessor	blend generation the coprocessor bit. It is cleared when the blend generation is finished.
10	mat_the_coprocessor	matrix multiplication the coprocessor bit. It the coprocessors the matrix multiplication state machine inside the MDPII. It is cleared by that state machine when it sees the po_final signal asserted. (Only valid for Matrix Multiplication instruction.)
11	int_the_coprocessor	interpolation the coprocessor bit. It the coprocessors the interpolation state machine inside MDPII. Once the coprocessor is set, this bit is set until the state machine sees po_final is asserted. (only valid for horizontal and interpolation instruction)
12	int_size	0 = interpolate with packed bytes 1 = interpolate with unpacked bytes or pixels (only valid for horizontal and vertical interpolation)
13	int_4	0 = interpolate by a factor of 2 1 = interpolate by a factor of 4 (only valid for horizontal and vertical interpolation)
14	int_vertical	0 = horizontal interpolation 1 = vertical interpolation
15	lock_step	0 = the PO, OOB and OOC interfaces are operating independent of each other 1 = the PO, OOB and OOC interfaces are locked together, so they will only accept data together.
19:16	reserved	
21:20	cw_config[1:0]	Clamp-or-wrapper configuration 0 = wrapped and no absolute value 1 = wrapped and absolute value 2 = clamp (overflow to 0xFF, underflow to 0x00), but no absolute value 3 = absolute value and clamp
22	fr_en	Fraction Rounder configuration 0 = disabled (returns 0) 1 = enabled

Table 1.112 mdp_op2 Register Fields

Field	Name	Description
24:23	oob_mode[1:0]	Mode of OOB interface operation: 0 = disabled 1 = sequential 2 = random 3 = blend_generation
26:25	ooc_mode[1:0]	Mode of OOC interface operation: 0 = disabled 1 = sequential 2 = GCSC 3 = pixel
30:27	trans[3:0]	In Compositing operation 0 = do not subtract offset for this channel 1 = subtract offset for this channel In Colour Space Conversion operation 0 = do not apply translation and clamping to output value on this channel 1 = use translation and clamping on this output channel In Image Transformation or convolution operations 0 = initialise accumulator to 0 for this channel 1 = initialise accumulator to mdp_por.0000 for this channel
31		reserved

dh. mdp_por

All bits are reset to zero.

Table 1.113 mdp_por Register Fields

Field	Name	Description
[7:0]	offset0[7:0]	offset for plus operator on channel 0
[15:8]	offset1[7:0]	offset for plus operator on channel 1
[23:16]	offset2[7:0]	offset for plus operator on channel 2
[31:24]	offset3[7:0]	offset for plus operator on channel 3
[31:0]	offset[31:0]	offset for convolutions and transformations

di. mdp_bi

All bits are reset to zero. The mdp_bi register is used for different things in different modes:

Table 1.114 mdp_bi Register Fields (compositing mode)

Field	Name	Description
[7:0]	blendend0	the coprocessor value of blend on channel 0
[15:8]	blendend1	the coprocessor value of blend on channel 1

246 B62

Table 1.114 mdp_bi Register Fields (compositing mode)

Field	Name	Description
[23:16]	blendend2	the coprocessor value of blend on channel 2
[31:24]	blendend3	the coprocessor value of blend on channel 3

Table 1.115 mdp_bi Register Fields (non-compositing mode)

Field	Name	Description
[31:2]	ioffset	offset into the index table

dj. mdp_bm

All bits are reset to zero. The mdp_bm register is used for different things in different modes:

Table 1.116 mdp_bm Register Fields (compositing mode)

Field	Field	Description
[7:0]	blendend0	end value of blend on channel 0
[15:8]	blendend1	end value of blend on channel 1
[23:16]	blendend2	end value of blend on channel 2
[31:24]	blendend3	end value of blend on channel 3

Table 1.117 mdp_bm Register Fields (non-compositing mode)

Field	Name	Description
[3:0]	rows[3:0]	number of rows in the matrix
[7:4]	cols[3:0]	number of columns in the matrix
[15:8]	level[7:0]	number of levels in halftoning
[20:16]	bp[4:0]	location of binary point

dk. mdp_len

All bits are reset to zero.

Table 1.118 mdp_len Register Fields

Field	Name	Description
31:0	length	length of blend to be produced

JPEG Coder Registers

dl. jc_cfg

Table 1.119 jc_cfg Register Fields for JPEG instructions

Field (Bits)	Name	Description
[31:28]	mop	Major opcode from instruction = 0010
[27]	D	0 = JPEG compress 1 = JPEG decompress
[26]	M	0 = single colour channel 1 = multiple colour channels
[25]	4	0 = three channel 1 = four channel
[24]	S	0 = do not use subsampling regime 1 = use subsampling regime
[23]		reserved
[22]	H	0 = use fast huffman algorithm 1 = use slow huffman algorithm
[21:16]		reserved
[15]	O	0 = JC is not operational 1 = JC is operational
[14]	dec	0 = disable decoding of instruction 1 = enable decoding of instruction
[13:8]		reserved
[7]	A	0 = align rethe coprocessort markers to byte boundaries 1 = align rethe coprocessort markers to word boundaries
[6]	Z	0 = pad with 1s 1 = pad with 0s
[5:4]		reserved
[3]	T	0 = not in self test mode 1 = in self test mode
[2]	B	0 = not bypassed 1 = bypassed
[1]	E	0 = disabled 1 = enabled
[0]		reserved

Table 1.120 jc_cfg Register Fields for data coding instructions

Field (Bits)	Name	Description
[31:28]	mop	Major opcode from instruction = 0010
[27]	D	0 = compress 1 = decompress
[26]		reserved
[25:24]	diff	Difference between the number of input bytes and the number of output bytes: 00 = no difference 01 = one extra output byte than input byte 10 = 11 = one less input byte than output byte
[23]	op	operation: 0 = huffman 1 = predictive code
[22:19]		reserved
[18:16]	ibo	input bit offset
[15]	O	0 = JC is not operational 1 = JC is operational
[14]	dec	0 = disable decoding of instruction 1 = enable decoding of instruction
[13:7]		reserved
[6]	Z	0 = pad with 1s 1 = pad with 0s
[5:4]		reserved
[3]	T	0 = not in self test mode 1 = in self test mode
[2]	B	0 = not bypassed 1 = bypassed
[1]	E	0 = disabled 1 = enabled
[0]		reserved

Table 1.121 jc_cfg Register Fields for memory copy instructions

Field (Bits)	Name	Description
[31:28]	mop	Major opcode from instruction = 1001
[27]	D	0 = general purpose data movement 1 = local DMA <i>This bit should always be set to '0'</i>
[26]	B	0 = not a bit copy operation 1 = bit copy operation

Table 1.121 jc_cfg Register Fields for memory copy instructions

Field (Bits)	Name	Description
[25:24]	diff	Difference between the number of input bytes and the number of output bytes: 00 = no difference 01 = one extra output byte than input byte 10 = 11 = one less input byte than output byte
[23]		reserved
[22:20]	obo	output bit offset
[19]		reserved
[18:16]	ibo	input bit offset
[15]	O	0 = JC is not operational 1 = JC is operational
[14]	dec	0 = disable decoding of instruction 1 = enable decoding of instruction
[13:4]		reserved
[3]	T	0 = not in self test mode 1 = in self test mode
[2]	B	0 = not bypassed 1 = bypassed
[1]	E	0 = disabled 1 = enabled
[0]		reserved

dm. jc_stat

Table 1.122 jc_stat Register Fields

Field (Bits)	Name	Description
[31:8]	reserved	
[7:0]	state	to be finalised

dn. jc_err_int

Table 1.123 jc_err_int Register Fields

Field (Bits)	Name	Description
[31:19]		reserved
[18]	huff_ill_tables	illegal huffman table. More than nine huffman table heap misses occurred.
[17]	huff_ill_mpos	illegal marker position
[16]	huff_ill_marker	illegal marker

Table 1.123 jc_err_int Register Fields

Field (Bits)	Name	Description
[15]	coeff_ill_AC	illegal AC coefficient value (-1024) during
[14]	coeff_ill_DC	illegal DC value
[13]	coeff_ill_AC_mag	illegal AC magnitude category
[12]	coeff_ill_DC_mag	illegal DC magnitude category
[11]	coeff_ill_RST	illegal RST _m count value
[10]	coeff_overflow	data overflow detected by coeff coder
[9]	coeff_ill_marker_pos	illegal marker position
[8]	jpeg_underflow	underflow
[1]	jpeg_disabled	received data while disabled error
[0]	huff_ill_symbol	illegal huffman symbol error

do. jc_err_int_en

Table 1.124 jc_err_int_en Register Fields

Field (Bits)	Name	Description
[31:19]		reserved
[18]	huff_ill_tables	illegal huffman table. More than nine huffman table heap misses occurred.
[17]	huff_ill_markers	illegal marker position
[16]	huff_ill_marker	illegal marker
[15]	coeff_ill_AC	illegal AC coefficient value (-1024) during
[14]	coeff_ill_DC	illegal DC value
[13]	coeff_ill_AC_mag	illegal AC magnitude category
[12]	coeff_ill_DC_mag	illegal DC magnitude category
[11]	coeff_ill_RST	illegal RST _m count value
[10]	coeff_overflow	data overflow detected by coeff coder
[9]	coeff_ill_marker_pos	illegal marker position
[8]	jpeg_underflow	underflow

Table 1.124 jc_err_int_cn Register Fields

Field (Bits)	Name	Description
[1]	jpeg_disable_d	received data while disabled error
[0]	huff_illegal_symbol	illegal huffman symbol error

dp. jc_rsi

Table 1.125 jc_rsi Register Fields

Field (Bits)	Name	Description
[15:0]	rsi	number of MCU blocks between the coprocessor markers

dq. jc_decode

Table 1.126 jc_decode Register Fields

Field (Bits)	Name	Description
[0]	dct_enable	enable dct submodule
[1]	dct_bypass	put dct submodule into bypass mode
[2]	dct_forward	put dct into forward mode
[3]	qdq_enable	enable quantizer submodule
[4]	qdq_bypass	bypass quantizer submodule
[5]	qdq_forward	put quantizer into forward mode
[6]	qdq_four	four channel image
[7]	qdq_subsmpl	subsampling image
[8]	cc_enable	enable the coeff coder submodule
[9]	cc_bypass	bypass coeff coder submodule
[10]	cc_forward	put coeff coder into forward mode
[11]	cc_jpeg	code jpeg compliant stream
[12]	cc_subsmpl	
[13]	cc_fourchannel	
[14]	cc_multichannel	
[15]	hc_enable	enable huffman coder submodule
[16]	hc_bypass	bypass huffman coder
[17]	hc_forward	put huffman coder into forward mode
[18]	hc_subsmpl	subsampling image
[19]	hc_fast	perform fast huffman coding
[20]	hc_jpeg	perform jpeg compliant huffman coding
[21]	hc_four	four channel image

Table 1.126 jc_decode Register Fields

Field (Bits)	Name	Description
[22]	hc_align	align RST _m markers on word boundaries
[23]	hc_zeropad	0 = pad using '1's 1 = pad using '0's
[24]	hc_memcpy	perform memory copy operation
[25]	misc_forward	
[26]	qdq_multi	
[27]	hc_multi	
[31:22]		reserved

dr. jc_res

Table 1.127 jc_res Register Fields

Field (Bits)	Name	Description
[7:0]	res	Residual value

ds. jc_table_sel

Table 1.128 jc_table_sel Register Fields

Field (Bits)	Name	Description
[13:8]	jc_table_sel_quant	decoded table selection for quantisation
[5:0]	jc_table_sel_huff	decoded table selection for huffman decoding

Result Organiser Registers

dt. ro_cfg

Table 1.129 ro_cfg Register Fields

Field	Name	Description
[31:23]	reserved	
[22:21]	chan	JPEG decompressed output data format : 0, 1 = Single-channel 2 = 3-channel 3 = 4-channel
[20]	upsample	0 = no upsampling 1 = upsample data from MUV RAM

253

Table 1.129 ro_cfg Register Fields

Field	Name	Description
[19:18]	reserved	
[17]	use_cut	0=do not use value in cut register to cut output data. 1=use value in cut register to cut output data. This bit is reset back to 0 on completion of an instruction.
[16]	use_limit	0=do not use value in limit register to limit output data. 1=use value in limit register to limit output data. This bit is reset back to 0 on completion of an instruction.
[15:14]		Reserved
[13:12]	mode[1:0]	Current mode of operation: 0 = idle 1 = sequential 2 = JPEG decompression 3 = CBus data
[11:4]		Reserved
[3]	S	0 = not in self test mode 1 = self test mode
[2]	B	0 = not bypassed 1 = bypassed
[1]	E	0 = disabled 1 = enabled
[0]		Reserved

du. ro_stat

Table 1.130 ro_stat Register Fields

Field	Name	Description
[31:24]		Reserved
[23:16]	state[7:0]	internal state condition : bit 0 = RO stalled bit 1 = RO active bit 2 = FIFO full bit 3 = FIFO empty
[15:0]		Reserved

dv. ro_err_int

Table 1.131 ro_err_int Register Fields

Field	Name	Description
[7:0]	error	error condition : bit 0 = request when disabled bit 1 = both MDP and JPEG Coder active at same time bit 2 = illegal address error bit 3 = jpeg mode data overflow

dw. ro_err_int_en

Table 1.132 ro_err_int_en Register Fields

Field	Name	Description
[7:0]	err_mask	error condition mask - enable corresponding error

dx. ro_dmr

Table 1.133 ro_dmr Register Fields

Field	Name	Description
[31:30]	ls3	lane swap for byte 3 : 0 = from byte 0 1 = from byte 1 2 = from byte 2 3 = no swap
[29:28]	ls2	lane swap for byte 2 : 0 = from byte 0 1 = from byte 1 2 = no swap 3 = from byte 3
[27:26]	ls1	lane swap for byte 1 : 0 = from byte 0 1 = no swap 2 = from byte 2 3 = from byte 3
[25:24]	ls0	lane swap for byte 0 : 0 = no swap 1 = from byte 1 2 = from byte 2 3 = from byte 3

Table 1.133 ro_dmr Register Fields

Field	Name	Description
[23:20]	suben[3:0]	Substitution Enables : 0 = do perform substitution for this byte 1 = use value stored in ro_subst for this byte
[19:16]	wrmask	Write Masks : 0 = write out corresponding byte channel 1 = do not write out corresponding byte channel
[15]	cmbs	Choose most significant bits 0=choose least significant bits of a byte when performing denormalisation 1= choose most significant bits of a byte when performing denormalisation
[14:12]	normalise	Denormalisation factor : 0 = 1 bit data objects 1 = 2 bit data objects 2 = 4 bit data objects 3 = 8 bit data objects 4 to 7 = 16 bit data objects
[11:8]		Reserved
[7]	P	External Format : 0 = unpacked bytes 1 = packed stream
[6:5]	if	Internal Format : 0 = pixels 1 = unpacked bytes 2 = packed bytes 3 = other
[4:3]	cc	Channel Count : 0 = 4 active channels 1 = 1 active channels 2 = 2 active channels 3 = 3 active channels
[2:0]		Reserved

dy. ro_subst

Table 1.134 ro_subst Register Fields

Field	Name	Description
[31:0]	subst[31:0]	substitution value or data value for Cbus mode

dz. ro_cdp

Table 1.135 ro_cdp Register Fields

Field	Name	Description
[31:0]	cdp[31:0]	address of current data item

ea. ro_len

Table 1.136 ro_len Register Fields

Field	Name	Description
[31:0]	len[31:0]	Output Byte count

eb. ro_sa

Table 1.137 ro_sa Register Fields

Field	Name	Description
[31:0]	sa[31:0]	the coprocessor address

ec. ro_idr

Table 1.138 ro_idr Register Fields

Field	Name	Description
[31:0]	height[15:0] width[31:16]	height - 1, in lines, of current image width - 1, in pixels, of current image

ed. ro_vbase

Table 1.139 ro_vbase Register Fields

Field	Name	Description
[31:12]	vbase[31:12]	the coprocessor Virtual Address Base

ee. ro_cut

Table 1.140 ro_cut Register Fields

Field	Name	Description
31:0	cut[31:0]	output cut offset: throw this many bytes away

el. ro_lmt

Table 1.141 ro_lmt Register Fields

Field	Name	Description
lmt[31:0]	lmt[31:0]	limit to the number of output bytes

PCI Configuration Space Alias

PCI configuration space is a 256-byte block of registers defined in the PCI spec, which allows the host to configure the PCI device, and to read its status. It is accessed using PCI configuration cycles. The register contents are also mirrored into a read-only area of the coprocessor's internal memory space, so that they can be read via normal PCI bus memory cycles.

The format of the configuration space implemented in the EIC is shown in Figure 1.1

Figure 1.1 the coprocessor PCI Configuration Space Layout

31		16 15		0	
Device ID		Vendor ID		0x00	
Status		Command		0x04	
Class Code			Revision ID		0x08
Reserved	Header Type	Latency Timer	Cache Line Size		0x0C
Base Address					0x10
Reserved					0x14-0x28
Subsystem ID		Subsystem Vendor ID		0x2C	
Reserved					0x30-0x38
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line		0x3C

Reserved registers, and reserved bits of implemented registers, will return 0 on reads and will not be affected by writes. Configuration space addresses in the range 0x40-0xFF are also reserved - no vendor specific configuration registers are defined.

eg. Vendor ID

This register is read only. CISRA's Vendor ID is 0x11AC.

eh. Device ID

This register is read only. the coprocessor's Device ID is 0x0001. The Device ID field is split up into two 8 bit fields: the most significant 8 bits is a number characteristic of the device (0x0 for the coprocessor) and the least significant 8 bits represents a version number of that device (0x1 for this version of the coprocessor).

ei. Command Register

The definitions of the fields in the Command register are shown in Table 1.142. All non-reserved bits in this register are read/write. After reset, this register is set to 0x0000.

Table 1.142 the coprocessor PCI Command Register Fields

Field(bits)	Name	Definition
0	IO Space	Reserved.
1	Memory Space	Enables the coprocessor to respond to memory cycles.
2	Bus Master	Enables the coprocessor to generate cycles on the PCI bus.
3	Special Cycles	Reserved.
4	Memory Write & Invalidate Enable	Reserved.
5	VGA Palette Snoop	Reserved.
6	Parity Error Response	Enables the assertion of <code>perr_1</code> on data parity errors, and <code>serr_1</code> on address parity errors (if bit 8 is also set). Also enables the setting of the Parity Error Detected bit in the Status register.
7	Wait cycle control	Reserved.
8	<code>serr_1</code> Enable	Enables the assertion of <code>serr_1</code> . the coprocessor will only assert <code>serr_1</code> on address parity errors.
9	Fast back-to-back Enable	Reserved.
10-15	Reserved	Reserved.

ej. Status Register

The definitions of the fields in the Status register are shown in Table 1.143. Reads to this register behave normally. Some bits of this register are read-only. The other bits can be set to 1 only by the coprocessor, and can be reset to 0 only by the host (except in test modes). The host resets a status bit by writing a 1 to that bit; writing a 0 to any bit has no effect. After reset, this register is set to 0x0280.

Table 1.143 the coprocessor PCI Status Register Fields

Field(bits)	Name	Definition
0-4	Reserved	Reserved
5	66MHz capable	Reserved.
6	User Definable Features	Reserved.
7	Fast Back-to-Back Capable	This bit is read-only, and set to 1. It indicates that the coprocessor behaves correctly in a system where fast back-to-back cycles are addressed to different target devices.

Table 1.143 the coprocessor PCI Status Register Fields

Field(bits)	Name	Definition
8	Data Parity Error Detected	This bit is set whenever the coprocessor, as a master, detects a data parity error in read data, or detects <code>perr_1</code> asserted on a write. It is only set if bit 6 of the command register is set. When this bit is set, an interrupt is signalled.
10-9	<code>devsel_1</code> timing	These read only bits are set to 01, to indicate that the coprocessor responds asserts its <code>devsel_1</code> with medium speed when it is a target.
11	Signalled Target Abort	Reserved.
12	Received Target Abort	This bit is set whenever the coprocessor, as a master, receives a target abort. It causes the EIC to generate an interrupt, and to stop its operation until the bit is cleared by software.
13	Received Master Abort	This bit is set whenever the coprocessor, as a master, executes a master abort. It causes the EIC to generate an interrupt, and to stop its operation until the bit is cleared by software.
14	Signalled System Error	This bit is set whenever the coprocessor asserts <code>serr_1</code> . This will only occur on an address parity error.
15	Detected Parity Error	This bit is set whenever the coprocessor detects an address or data parity error.

ek. Revision ID

This is a read-only register. the coprocessor's initial revision ID is 0x01.

el. Class Code

This is a read-only register. the coprocessor does not fit into any of the defined class codes of the PCI SIG, so this register is set to 0xFF0000.

em. Cache Line Size

This is a read/write register that specifies the cache line size of the system in 32 bit words. It determines when the coprocessor will use the Memory Read Line and Memory Read Multiple commands. the coprocessor supports any value from 0 to 255 in this register. A value of zero in this register disables the Memory Read Line and Memory Read Multiple command types. It is set to 0x00 on reset.

en. Latency Timer

This is a read/write register that specifies the largest number of clocks the coprocessor can use for any PCI transaction. the coprocessor supports any value in this register from 0 to 255. It is set to 0x00 on reset.

eo. Header Type

This read-only register is set to 0x00, indicating that the coprocessor uses a type 0 layout for the configuration space.

ep. Base Address

This read/write register is used to locate the coprocessor's internal registers and memories, the local memory and the Generic Interface in the host's memory map. the coprocessor's various resources occupy 64 MBytes (not all locations are used), so only the top 6 bits of this

register are writable. The remaining address bits are all hardwired to zero. The lower four bits of this register are read-only control bits, which are also hardwired to 0. This indicates that the register refers to memory space, that the coprocessor can be mapped anywhere in the 32-bit address space of the host, and that the coprocessor resources are not prefetchable when it is a target.

eq. Subsystem Vendor ID

This read-only register allows the host to identify the vendor of a PCI board plugged into the system (as opposed to the vendor of the component implementing the PCI interface on the board). The contents of this register are loaded on reset via the serial configuration port on the EIC.

er. Subsystem ID

The read-only register allows the host to identify a PCI board plugged into the system. The contents of this register are loaded on reset via the serial configuration port on the EIC. This mechanism allows any required information about the board functionality or configuration to be encoded externally and read by the host.

es. Interrupt Line

This read/write register is used by the system software to record interrupt line routing information so that it is accessible to interrupt service software. It has no effect on any operations in the coprocessor. It is set to 0x00 on reset.

et. Interrupt Pin

This read-only register is hard-wired to 0x01, indicating that the coprocessor drives PCI interrupt pin `inta_1`.

eu. Min_Gnt

This read only register indicates to the system the coprocessor's desired burst period length, in units of 1/4 microseconds. The optimum value for this register has not yet been determined.

ev. Max_Lat

This read only register indicates to the system the coprocessor's desired maximum latency in gaining control of the PCI bus after a request, in units of 1/4 microseconds. The optimum value for this register has not yet been determined.

1.1.4 Internal Memory Map

This section details the objects that appear in the per-module data areas within the the coprocessor internal memory map.

Table 1.144 the coprocessor Internal Memory Map

Byte Offset from the coprocessor base	Associated Module	Name	Type	Definition
0x0000-0x1FFF	reserved	reserved		
0x8000-0x803F	EIC	eic_ptp	R/W	MMU page table pointers
0x8040-0x807F	reserved	reserved		

Table 1.144 the coprocessor Internal Memory Map

Byte Offset from the coprocessor base	Associated Module	Name	Type	Definition
0x8080-0x80FF	EIC	eic_addr	Read Only	CAM data with virtual and physical page numbers interleaved
0x8100-0xFFFF	reserved	reserved		
0x10000-0x1FFFF	LMC	reserved		
0x20000-0x2007F	PIC	input RAM	R/W	32 word x32 bit RAM in PIC input datapath
0x20080-0x200FF	PIC	output RAM	R/W	32 word x32 bit RAM in PIC output datapath
0x20100-0x2FFFF	PIC	reserved		
0x30000-0x3FFFF	MISC	reserved		
0x40000-40017	IC	prefetch_buf	read only	Contents of Prefetch Buffer
0x40018-0x4FFFF	IC	reserved		
0x50000	DCC	cache RAM	R/W	4k byte of 32 bit wide memory
0x51000	DCC	tag RAM	R/W	128 x 20 bit tag memory - the Most significant 20 bits of the 32 bit word are used
0x52000	DCC	dcc_odatab0	R/W	bits 31:0 of o_datab bus. This value is driven on to the bus in the test mode.
0x52004	DCC	dcc_odatab1	R/W	bits 63:32 of o_datab bus. This value is driven on to the bus in the test mode.
0x52008	DCC	dcc_odatab2	R/W	bits 95:64 of o_datab bus. This value is driven on to the bus in the test mode.
0x5200c	DCC	dcc_odatab3	R/W	bits 127:96 of o_datab bus. This value is driven on to the bus in the test mode.
0x52010	DCC	dcc_odatac0	R/W	bits 31:0 of o_datac bus. This value is driven on to the bus in the test mode.
0x52014	DCC	dcc_odatac1	R/W	bits 63:32 of o_datac bus. This value is driven on to the bus in the test mode.
0x52018	DCC	dcc_odatac2	R/W	bits 95:64 of o_datac bus. This value is driven on to the bus in the test mode.

Table 1.144 the coprocessor Internal Memory Map

Byte Offset from the coprocessor base	Associated Module	Name	Type	Definition
0x5201c	DCC	dcc_odatac3	R/W	bits 127:96 of o_datac bus. This value is driven on to the bus in the test mode.
0x060000 - 0x0607FF	PO	mu_v_ram	R/W	MUV RAM. The MUV ram appears in the same format as the Interval and Fraction RAM.
0x060800 - 0x06FFFF	PO	reserved		
0x70000 - 0x7FFFF	OOB	reserved		
0x80000 - 0x8FFFF	OOC	reserved		
0x90000 - 0x900FC	JC	Quantizer Buffer 1		Uses only the least significant 15 bits. The most significant 17 bits are reserved.
0x90100 - 0x901FC	JC	Quantizer Buffer 2		Uses only the least significant 15 bits. The most significant 17 bits are reserved.
0x90200 - 0x902FC	JC	DCT Buiffer		Uses only the least significant 12 bits. The most significant 20 bits are reserved.
0x90300 - 0x9FFFF	JC	reserved		
0x0A0000 - 0x0A01FF	MDP			Lots of internal structures.
0x0A0200 - 0x0AFFFF	MDP	reserved		
0x0B0000 - 0x0BFFFF	RO	reserved		
0xC0000 - 0xCFFFF	-	reserved		
0xD0000 - 0xDFFFF	IIS	reserved		
0xE0000 - 0xFFFF	-	reserved		

263

331

1.1.5 Memory Word Fields

a. eic_ptp

Table 1.145 eic_ptp Memory Word Fields

Field(bits)	Name	Definition
11:0	Reserved	
31:12	ptp	Top 20 bits of physical byte address of the bottom of a 4kB segment of the page table.

4. BRIEF DESCRIPTION OF THE DRAWINGS

Notwithstanding any other forms which may fall within the scope of the present invention, preferred forms of the invention will now be described, by way of example only, with reference to the accompanying drawings:

Fig. 1 illustrates the operation of a raster image co-processor within a host computer environment;

Fig. 2 illustrates the raster image co-processor of Fig. 1 in further detail;

Fig. 3 illustrates the memory map of the raster image co-processor;

Fig. 4 shows the relationship between a CPU, instruction queue, instruction operands and results in shared memory, and a co-processor;

Fig. 5 shows the relationship between an instruction generator, memory manager, queue manager and co-processor;

Fig. 6 shows the operation of the graphics co-processor reading instructions for execution from the pending instruction queue and placing them on the completed instruction queue;

Fig. 7 shows a fixed length circular buffer implementation of the instruction queue, indicating the need to wait when the buffer fills;

Fig. 8 illustrates to instruction execution streams as utilized by the co-processor;

Fig. 9 illustrates an instruction execution flow chart;

Fig. 10 illustrates the standard instruction word format utilized by the co-processor;

Fig. 11 illustrates the instruction word fields of a standard instruction;

Fig. 12 illustrates the data word fields of a standard instruction;

Fig. 13 illustrates schematically the instruction controller of Fig. 2;

Fig. 14 illustrates the execution controller of Fig. 13 in more detail;

Fig. 15 illustrates a state transition diagram of the instruction controller;

Fig. 16 illustrates the instruction decoder of Fig. 13;

Fig. 17 illustrates the instruction sequencer of Fig. 16 in more detail;

Fig. 18 illustrates a transition diagram for the ID sequencer of Fig. 16;

Fig. 19 illustrates schematically the prefetch buffer controller of Fig. 13 in more detail;

Fig. 20 illustrates the standard form of register storage and module interaction as utilized in the co-processor;

Fig. 21 illustrates the format of control bus transactions as utilized in the co-processor;

5 Fig. 22 illustrates the data flow through a portion of the co-processor;

Figs. 23-29 illustrate various examples of data reformatting as utilized in the co-processor;

Figs. 30 and 31 illustrate the format conversions carried out by the co-processor;

10 Fig. 32 illustrates the process of input data transformation as carried out in the co-processor;

Figs. 33-41 illustrate various further data transformations as carried out by the co-processor;

Fig. 42 illustrates various internal to output data transformations carried out by the co-processor;

15 Figs. 43-47 illustrate various further example data transformations carried out by the co-processor;

Fig. 48 illustrates various fields utilized by internal registers to determine what data transformations should be carried out;

20 Fig. 49 depicts a block diagram of a graphics subsystem that uses data normalization.;

Fig. 50 illustrates a circuit diagram of a data normalization apparatus;

Fig. 51 illustrates the pixel processing carried out for compositing operations;

Fig. 52 illustrates the instruction word format for compositing operations;

Fig. 53 illustrates the data word format for compositing operations;

25 Fig. 54 illustrates the instruction word format for tiling operations;

Fig. 55 illustrates the operation of a tiling instruction on an image;

Fig. 56 illustrates the process of utilization of interval and fractional tables to re-map color gamuts;

30 Fig. 57 illustrates the form of storage of interval and fractional tables within the MUV buffer of the co-processor;

Fig. 58 illustrates the process of color conversion utilising interpolation as carried out in the co-processor;

Fig. 59 illustrates the refinements to the rest of the color conversion process at gamut edges as carried out by the co-processor;

Fig. 60 illustrates the process of color space conversion for one output color as implemented in the co-processor;

5 Fig. 61 illustrates the memory storage within a cache of the co-processor when utilising single color output color space conversion;

Fig. 62 illustrates the methodology utilized for multiple color space conversion;

Fig. 63 illustrates the process of address re-mapping for the cache when utilized during the process of multiple color space conversion;

10 Fig. 64 illustrates the instruction word format for color space conversion instructions;

Fig. 65 illustrates a method of multiple color conversion;

Fig. 66 and 67 illustrate the formation of MCU's during the process of JPEG conversion as carried out in the co-processor;

15 Fig. 68 illustrates the structure of the JPEG coder of the co-processor;

Fig. 69 illustrates the quantizer portion of Fig. 68 in more detail;

Fig. 70 illustrates the Huffman coder of Fig. 68 in more detail;

Figs. 71 and 72 illustrate the Huffman coder and decoder in more detail;

20 Figs. 73-75 illustrate the process of cutting and limiting of JPEG data as utilized in the co-processor;

Fig. 76 illustrates the instruction word format for JPEG instructions;

Fig. 77 shows a block diagram of a typical discrete cosine transform apparatus (prior art);

Fig. 78 illustrates an arithmetic data path of a prior art DCT apparatus;

25 Fig. 79 shows a block diagram of a DCT apparatus utilized in the co-processor;

Fig. 80 depicts a block diagram of the arithmetic circuit of Fig. 79 in more detail;

Fig. 81 illustrates an arithmetic data path of the DCT apparatus of Fig. 79;

Fig. 82 presents a representational stream of Huffman-encoded data units interleaved with not encoded bit fields, both byte aligned and not, as in JPEG format;

30 Fig. 83 illustrates the overall architecture of a Huffman decoder of JPEG data of Fig. 84 in more detail;

Fig. 84 illustrates the overall architecture of the Huffman decoder of JPEG data;

Fig. 85 illustrates data processing in the stripper block which removes byte aligned not encoded bit fields from the input data. Examples of the coding of tags corresponding to the data outputted by the stripper are also shown;

Fig. 86 shows the organization and the data flow in the data preshifter;

5 Fig. 87 shows control logic for the decoder of Fig. 81;

Fig. 88 shows the organization and the data flow in the marker preshifter;

Fig. 89 shows a block diagram of a combinatorial unit decoding Huffman encoded values in JPEG context;

10 Fig. 90 illustrates the concept of a padding zone and a block diagram of the decoder of padding bits;

Fig. 91 shows an example of a format of data outputted by the decoder, the format being used in the co-processor;

Fig. 92 illustrates methodology utilized in image transformation instructions;

15 Fig. 93 illustrates the instruction word format for image transformation instructions;

Figs 94 and 95 illustrate the format of an image transformation kernel as utilized in the co-processor;

Fig. 96 illustrates the process of utilising an index table for image transformations as utilized in the co-processor;

20 Fig. 97 illustrates the data field format for instructions utilising transformations and convolutions;

Fig. 98 illustrates the process of interpretation of the bp field of instruction words;

Fig. 99 illustrates the process of convolution as utilized in the co-processor;

25 Fig. 100 illustrates the instruction word format for convolution instructions as utilized in the co-processor;

Fig. 101 illustrates the instruction word format for matrix multiplication as utilized in the co-processor;

30 Figs 102-105 illustrates the process utilized for hierarchial image manipulation as utilized in the co-processor;

Fig. 106 illustrates the instruction word coding for hierarchial image instructions;

Fig. 107 illustrates the instruction word coding for flow control instructions as illustrated in the co-processor;

Fig. 108 illustrates the pixel organizer in more detail;

Fig. 109 illustrates the operand fetch unit of the pixel organizer in more detail;

Figs. 110-114 illustrate various storage formats as utilized by the co-processor;

Fig. 115 illustrates the MUV address generator of the pixel organizer of the co-processor in more detail;

Fig. 116 is a block diagram of a multiple value (MUV) buffer utilized in the co-processor;

Fig. 117 illustrates a structure of the encoder of Fig. 116;

Fig. 118 illustrates a structure of the decoder of Fig. 116;

Fig. 119 illustrates a structure of an address generator of Fig. 116 for generating read addresses when in JPEG mode (pixel decomposition);

Fig. 120 illustrates a structure of an address generator of Fig. 116 for generating read addresses when in JPEG mode (pixel reconstruction);

Fig. 121 illustrates an organization of memory modules comprising the storage device of Fig 116;

Fig. 122 illustrates a structure of a circuit that multiplexes read addresses to memory modules;

Fig. 123 illustrates a representation of how lookup table entries are stored in the buffer operating in a single lookup table mode;

Fig. 124 illustrates a representation of how lookup table entries are stored in the buffer operating in a multiple lookup table mode;

Fig. 125 illustrates a representation of how pixels are stored in the buffer operating in JPEG mode (pixel decomposition);

Fig. 126 illustrate a representation of how single color data blocks are retrieved from the buffer operating in JPEG mode (pixel reconstruction);

Fig. 127 illustrates the structure of the result organizer of the co-processor in more detail;

Fig. 128 illustrates the structure of the operand organizers of the co-processor in more detail;

Fig. 129 is a block diagram of a computer architecture for the main data path unit utilized in the co-processor;

Fig. 130 is a block diagram of a input interface for accepting, storing and rearranging input data objects for further processing;

Fig. 131 is a block diagram of a image data processor for performing arithmetic operations on incoming data objects;

Fig. 132 is a block diagram of a color channel processor for performing arithmetic operations on one channel of the incoming data objects;

5 Fig. 133 is a block diagram of a multifunction block in a color channel processor;

Fig. 134 illustrates a block diagram for compositing operations;

Fig. 135 shows an inverse transform of the scanline;

Fig. 136 shows a block diagram of the steps required to calculate the value for a designation pixel;

10 Fig. 137 illustrates a block diagram of the image transformation engine;

Fig. 138 illustrates the two formats of kernel descriptions;

Fig. 139 shows the definition and interpretation of a bp field;

Fig. 140 shows a block diagram of multiplier-adders that perform matrix multiplication;

15 Fig. 141 illustrates the control, address and data flow of the cache and cache controller of the co-processor;

Fig. 142 illustrates the memory organization of the cache;

Fig. 143 illustrates the address format for the cache controller of the co-processor;

20 Fig. 144 is a block diagram of a multifunction block in a color channel processor;

Fig. 145 illustrates the input interface switch of the co-processor in more Fig. 144 illustrates, a block diagram of the cache and cache controller;

Fig. 146 illustrates a four-port dynamic local memory controller of the co-processor showing the main address and data paths;

25 Fig. 147 illustrates a state machine diagram for the controller of Fig. 146;

Fig. 148 is a pseudo code listing detailing the function of the arbitrator of Fig. 146;

Fig. 149 depicts the structure of the requester priority bits and the terminology used in Fig. 146.

30 Fig. 150 illustrates the external interface controller of the co-processor in more detail;

Figs. 151-154 illustrate the process of virtual to/from physical address mapping as utilized by the co-processor;

358

- 270 -

Fig. 155 illustrates the IBus receiver unit of Fig. 150 in more detail;
Fig. 156 illustrates the RBus receiver unit of Fig. 2 in more detail;
Fig. 157 illustrates the memory management unit of Fig. 150 in more detail;
Fig. 158 illustrates the peripheral interface controller of Fig. 2 in more detail.

5

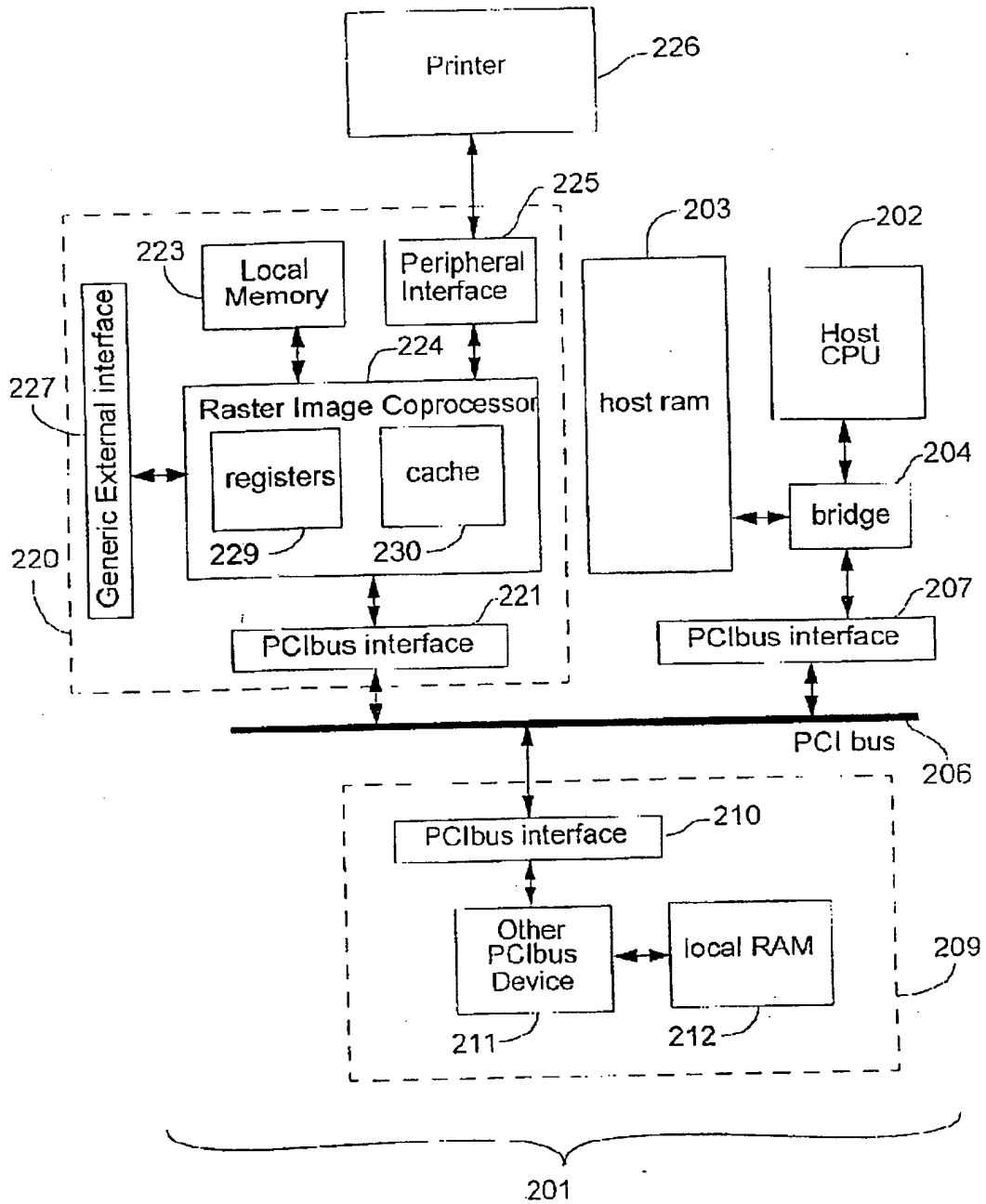


Fig. 1

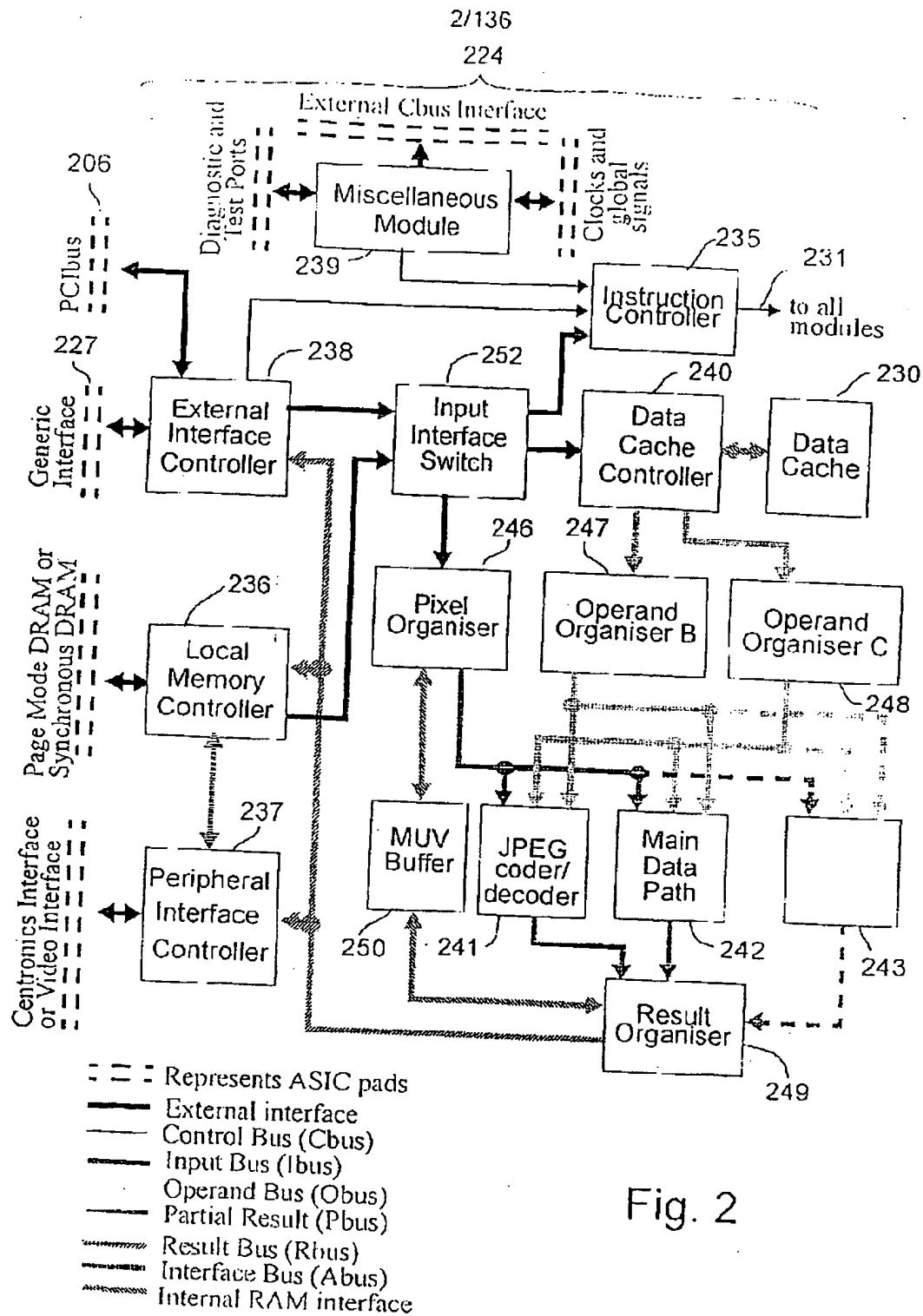


Fig. 2

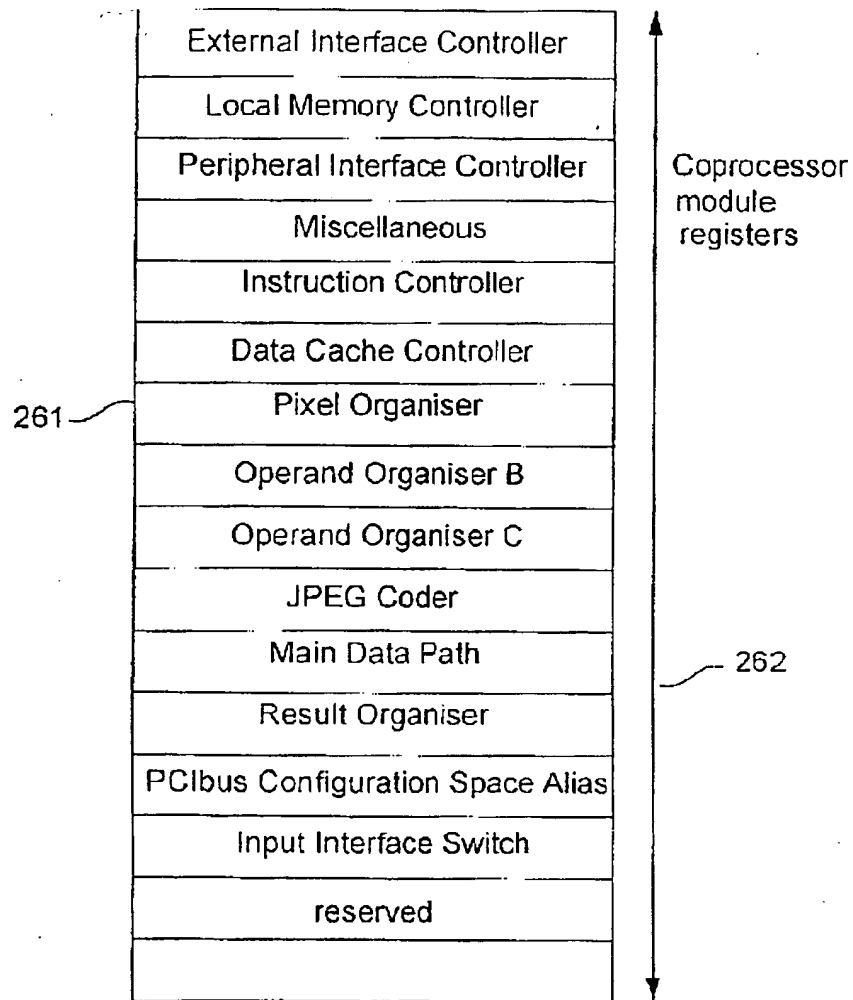


Fig. 3

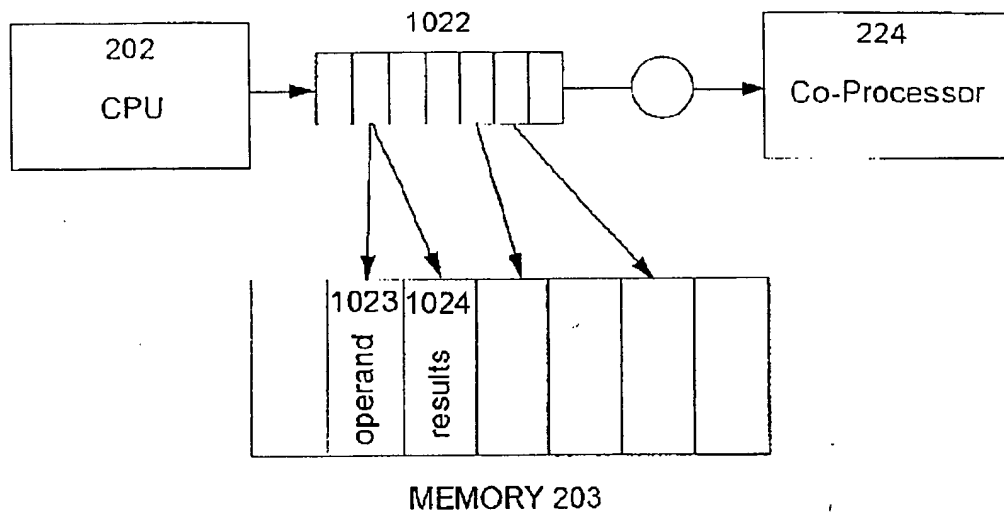


Fig. 4

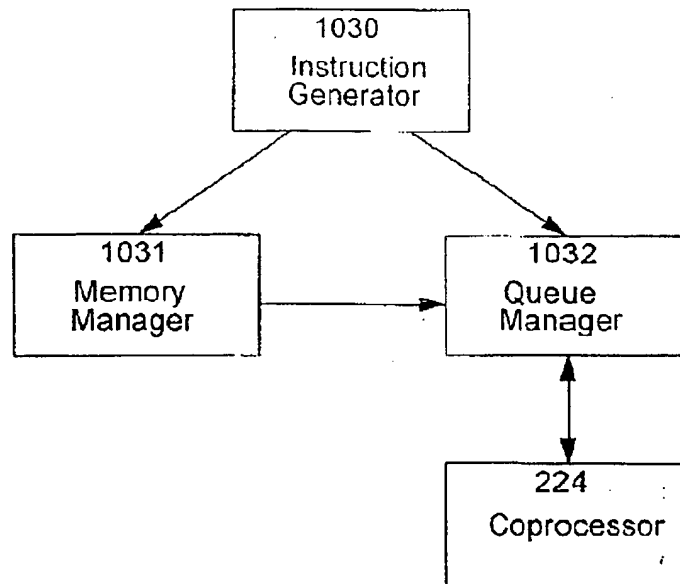


Fig. 5

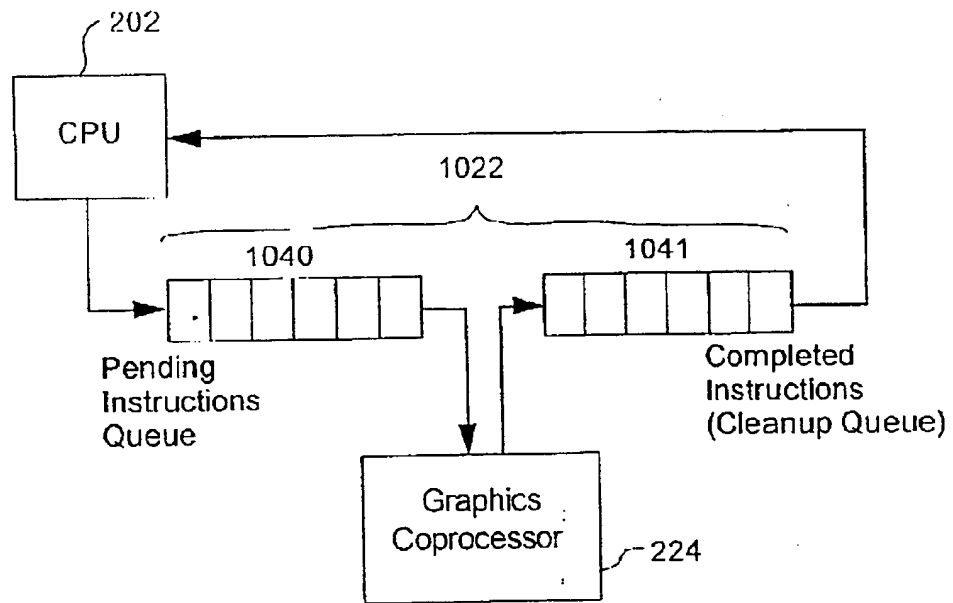


Fig. 6

Queue Manager

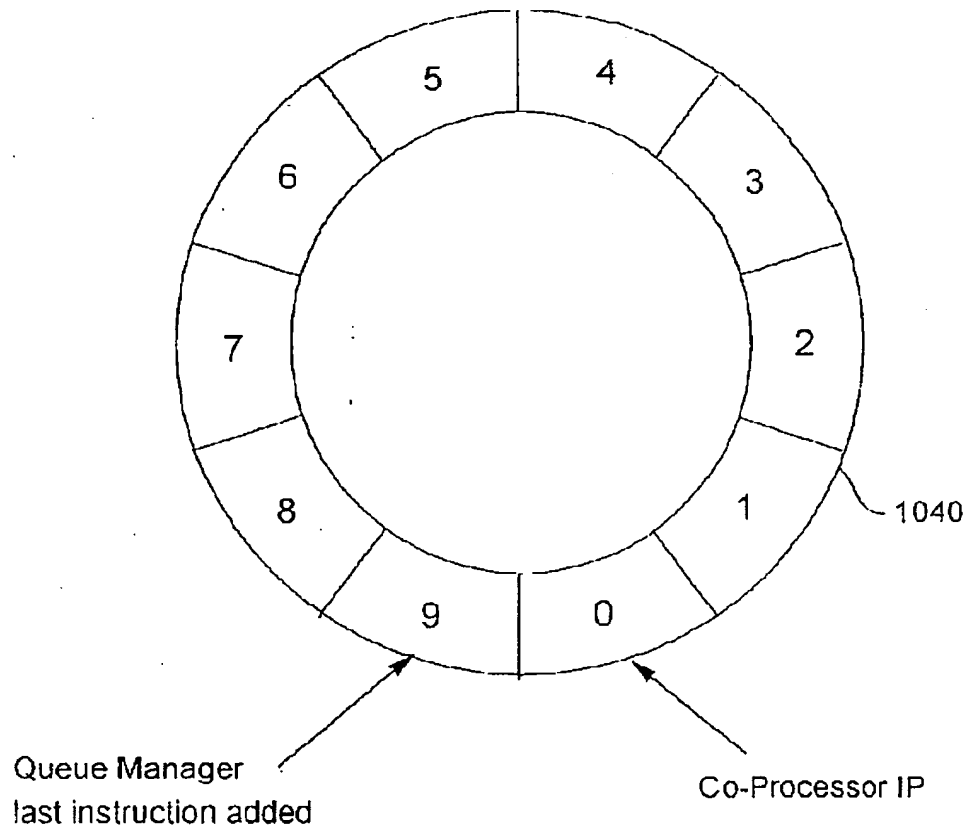


Fig. 7

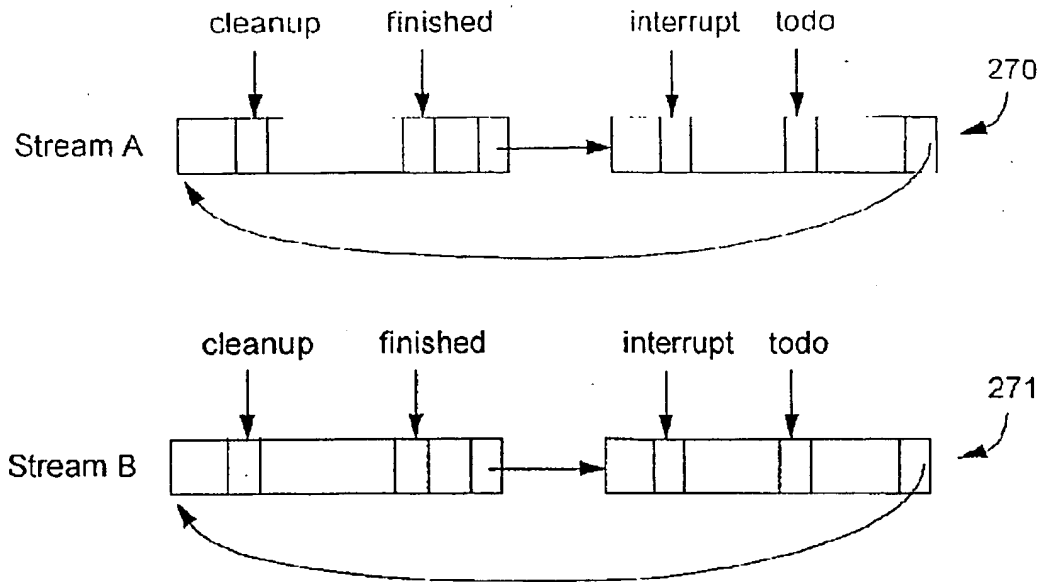


Fig. 8

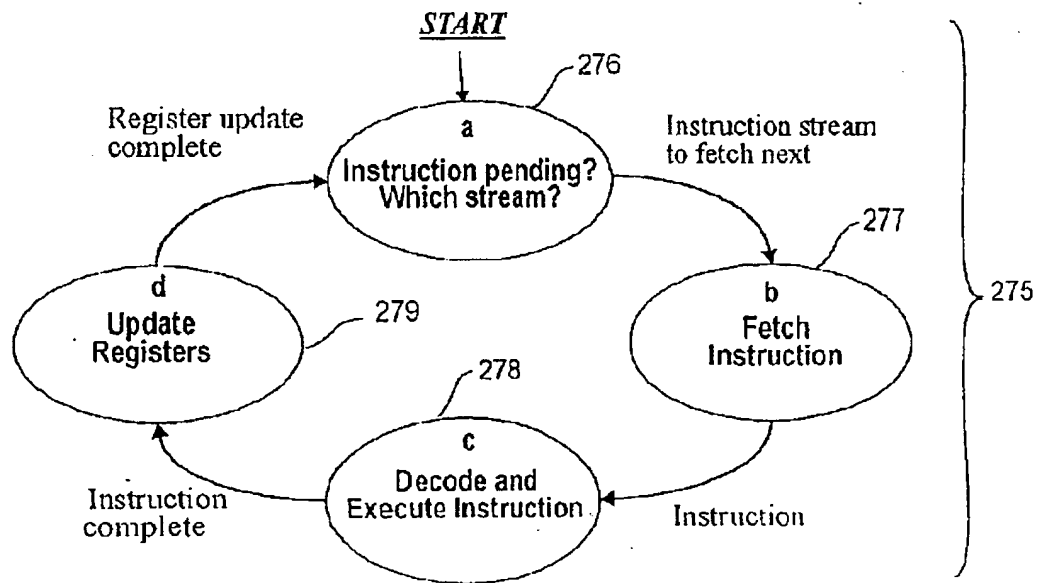


Fig. 9

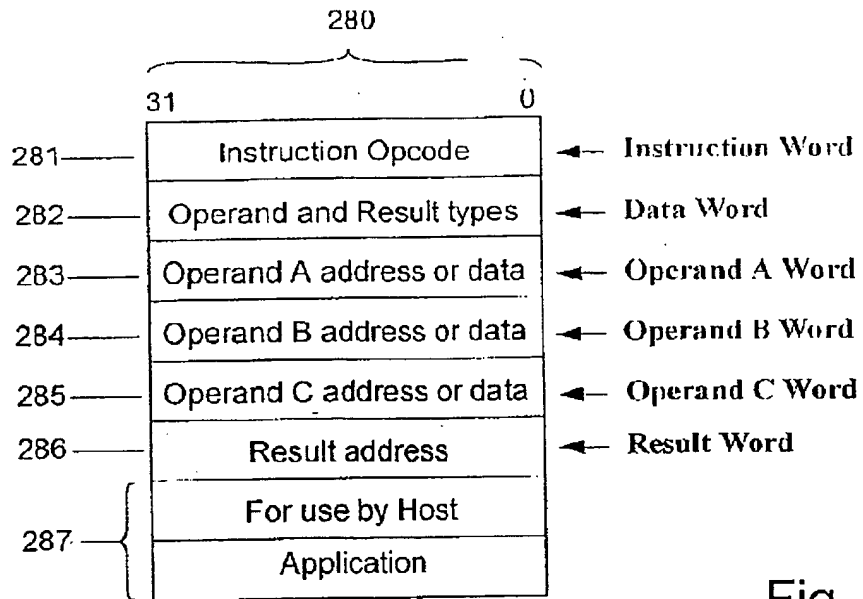


Fig. 10

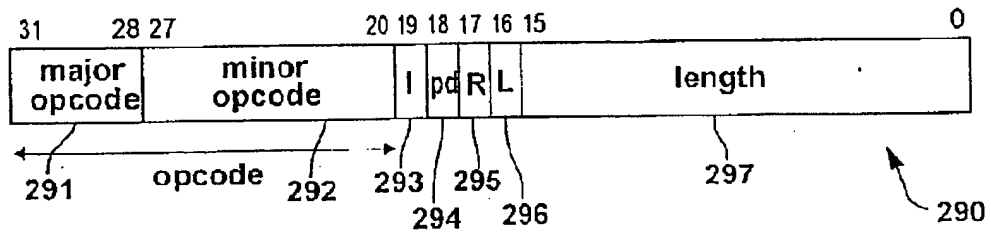
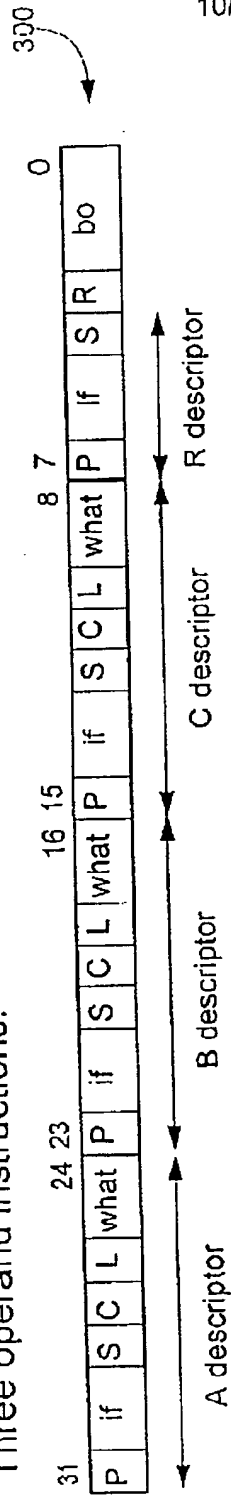


Fig. 11

Format of the Data Word

Three operand instructions:



10/136

Two (or less) operand instructions:

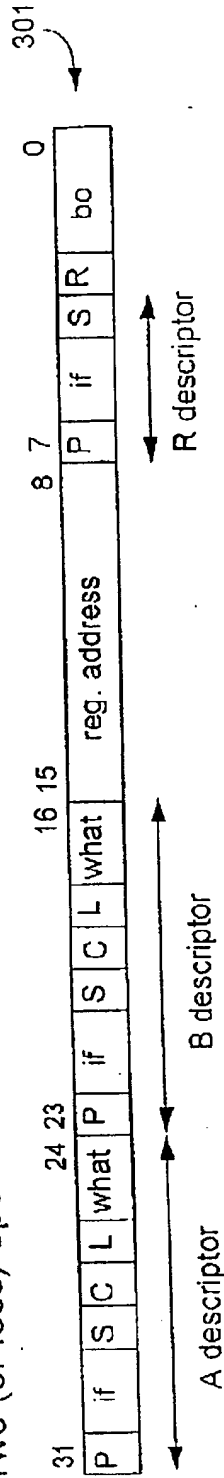


Fig. 12

235

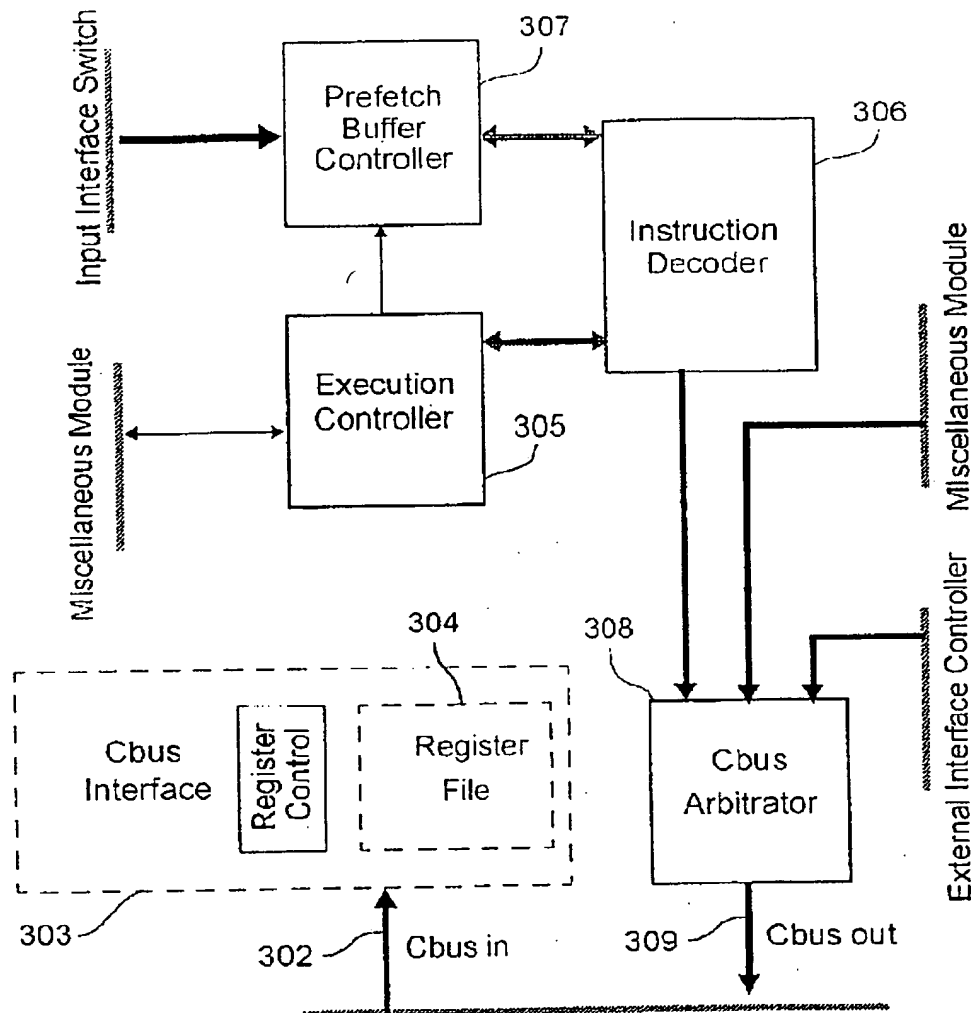


Fig. 13

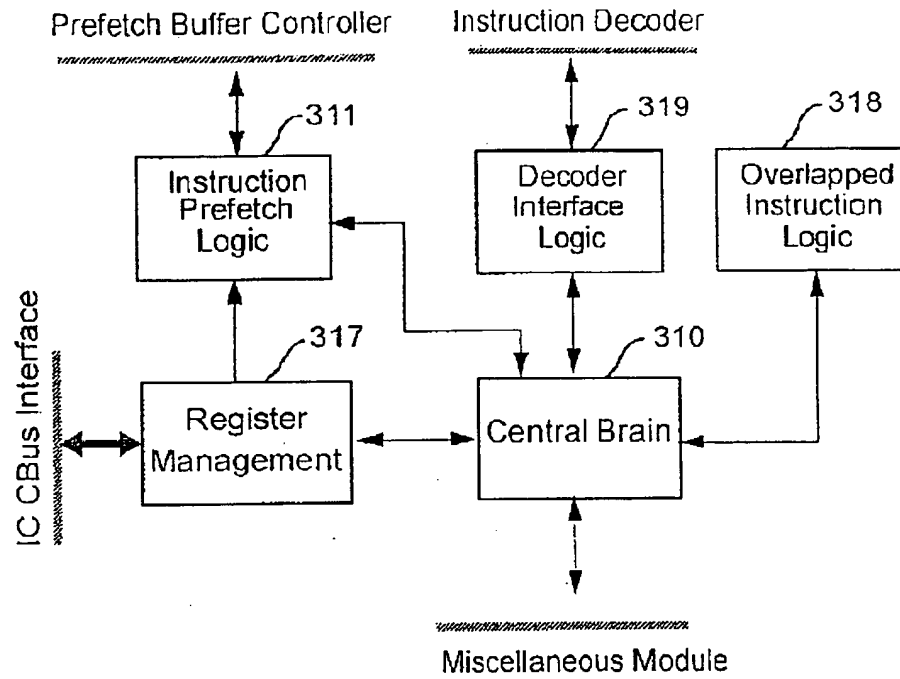


Fig. 14

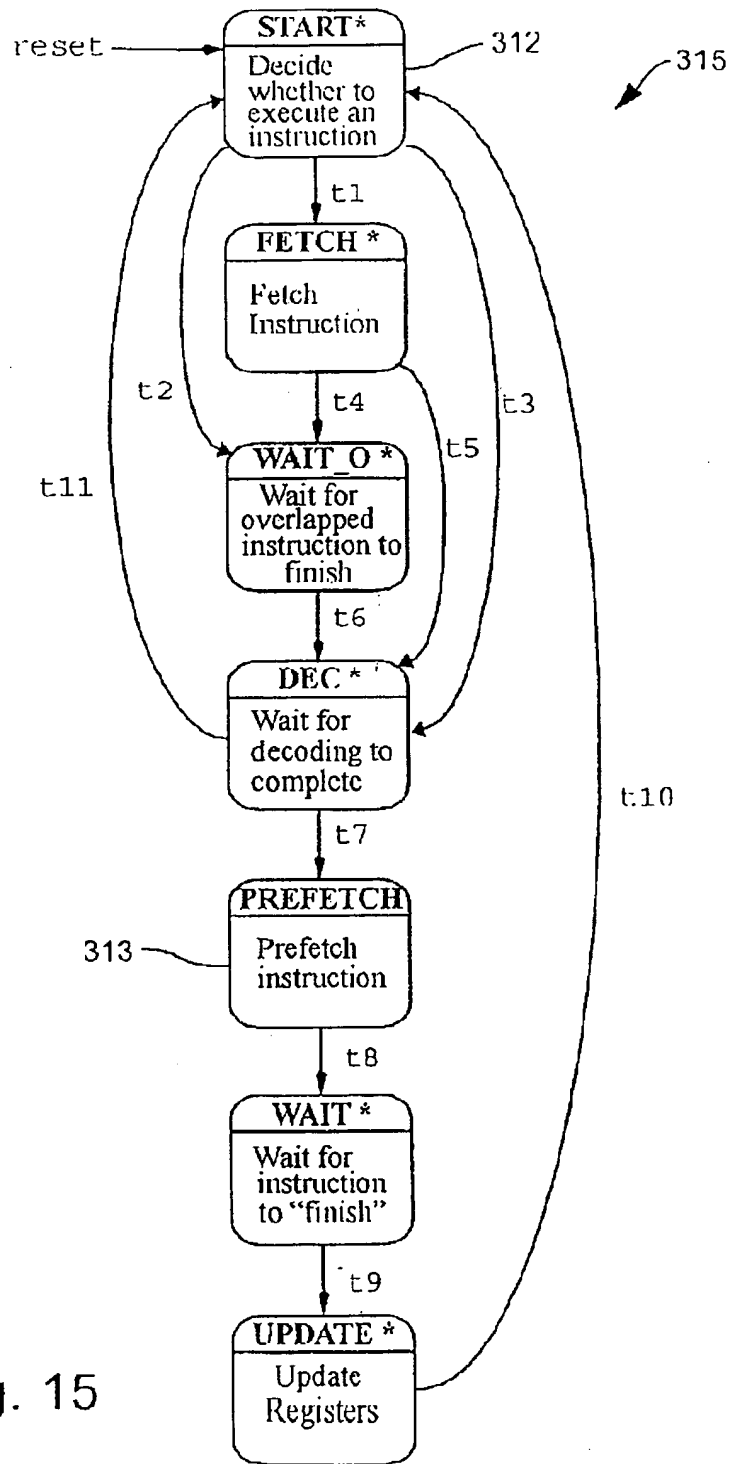


Fig. 15

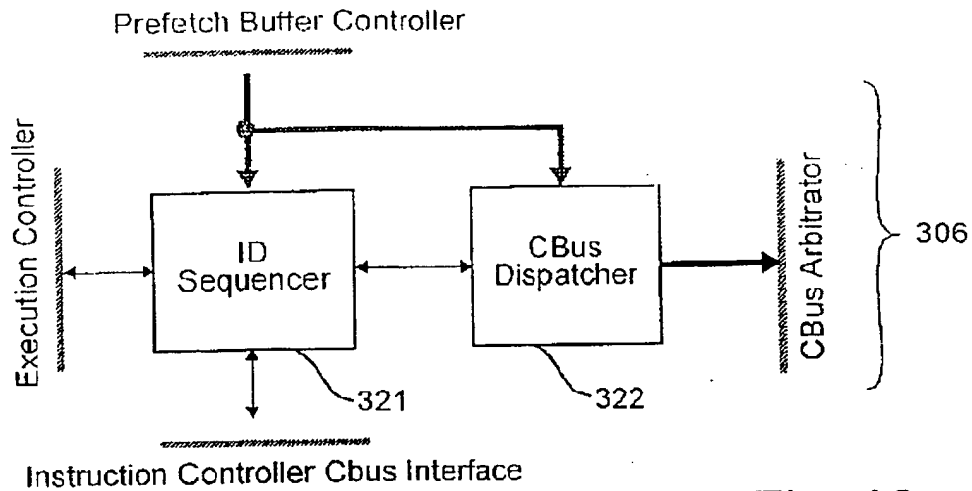


Fig. 16

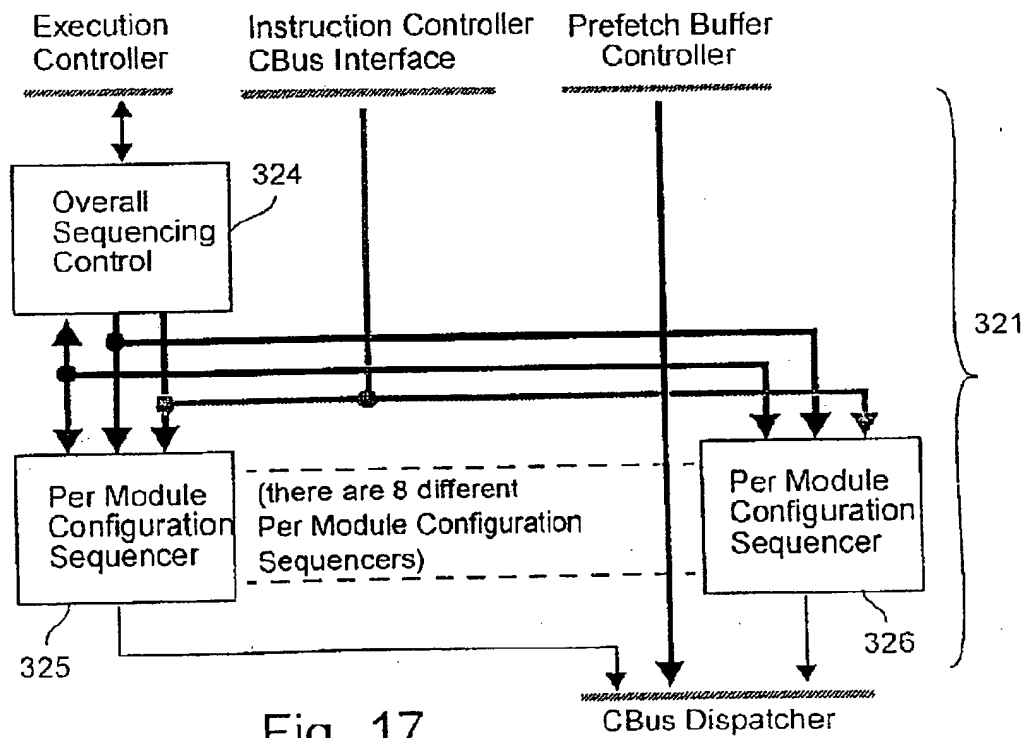


Fig. 17

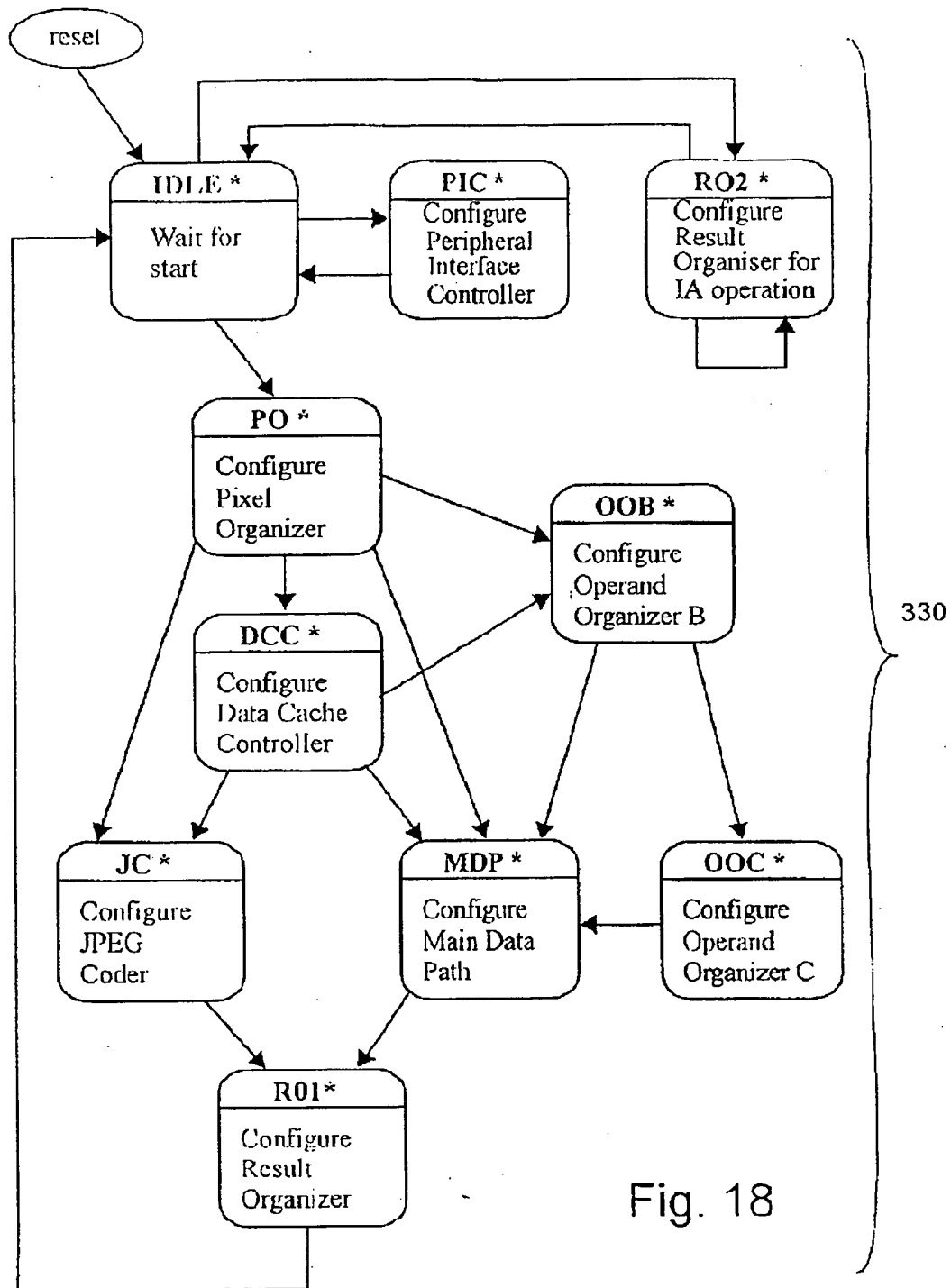


Fig. 18

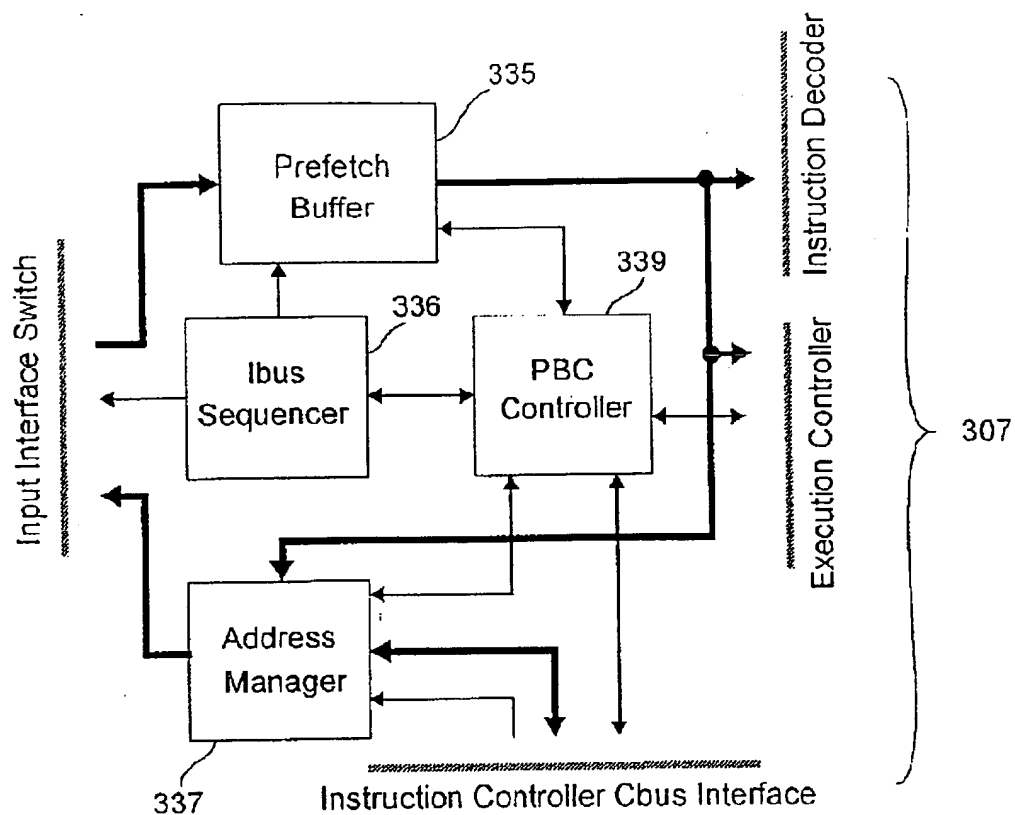


Fig. 19

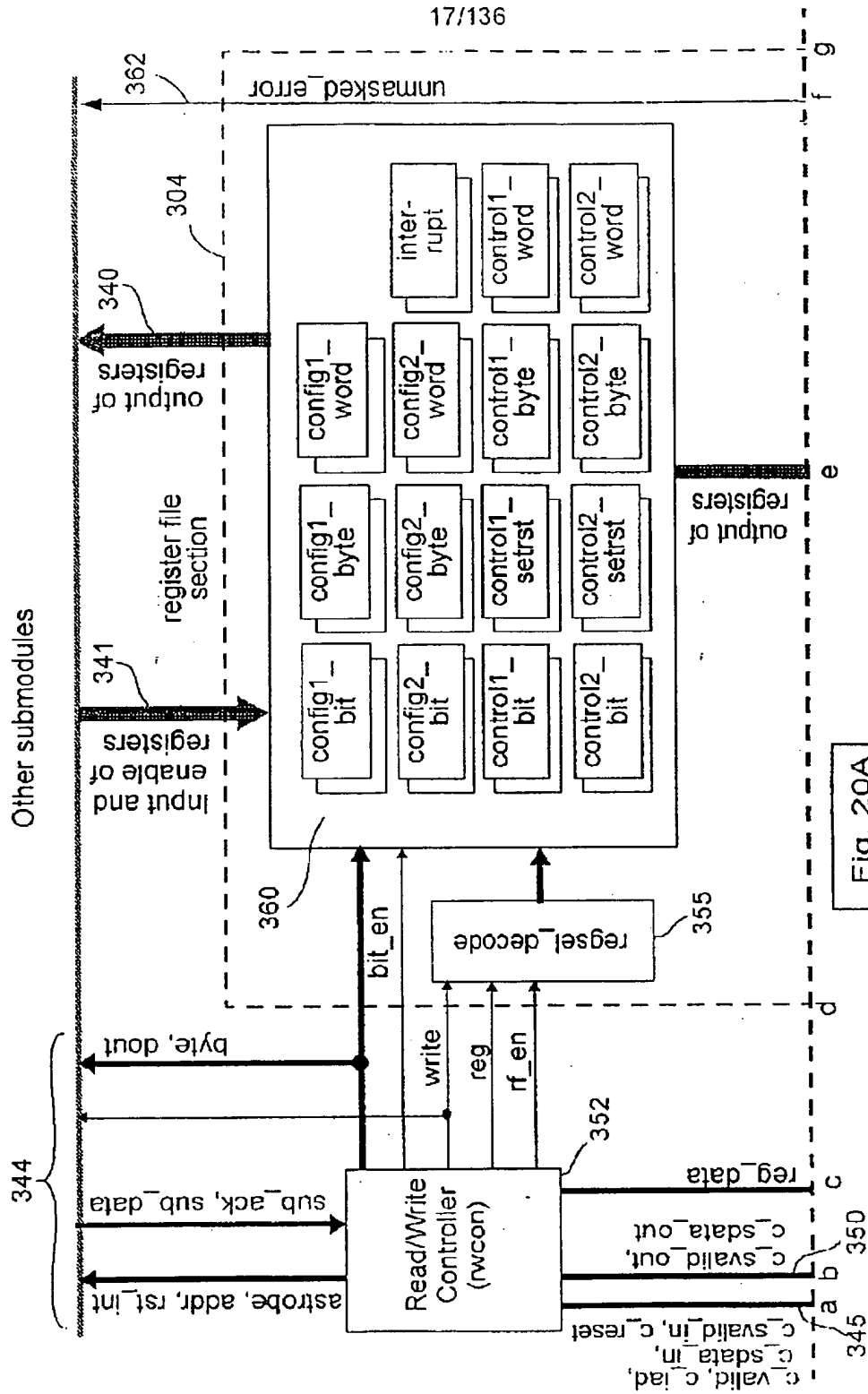


Fig. 20A

Fig. 20A
Fig. 20B

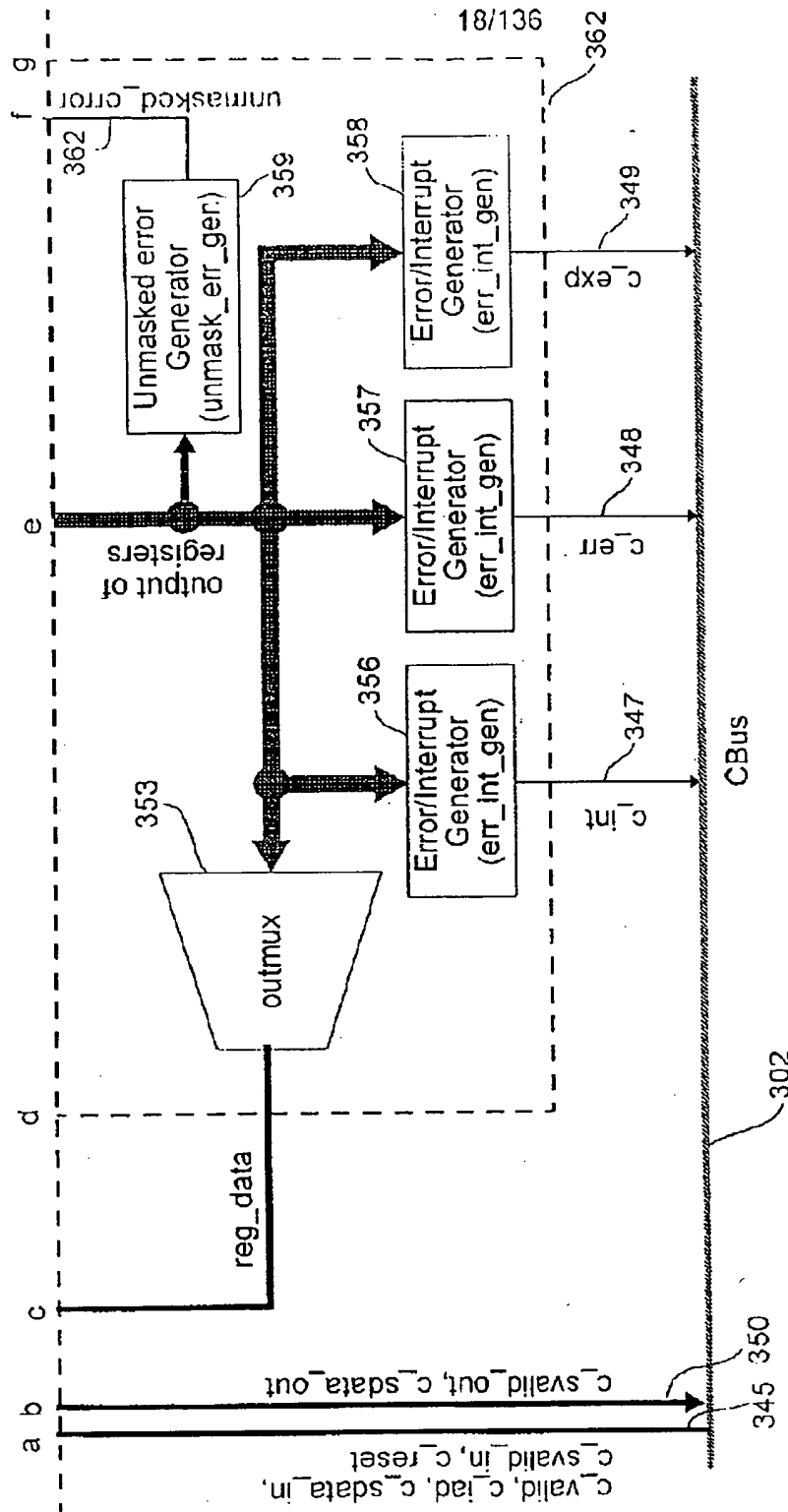
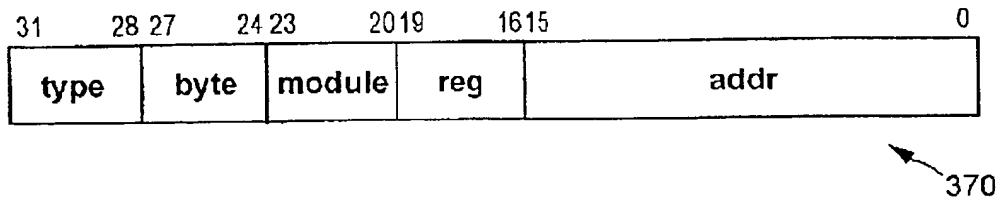


Fig. 20B

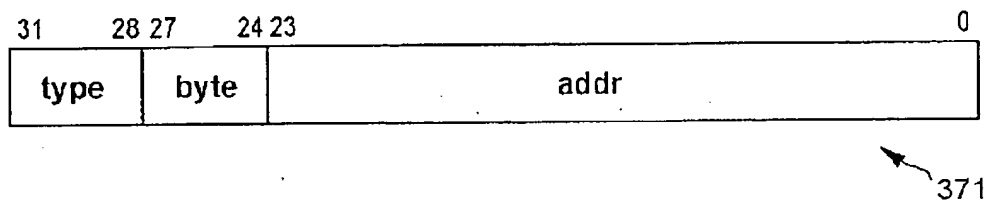
Fig. 20A

Fig. 20B

Type A



Type B



Type C

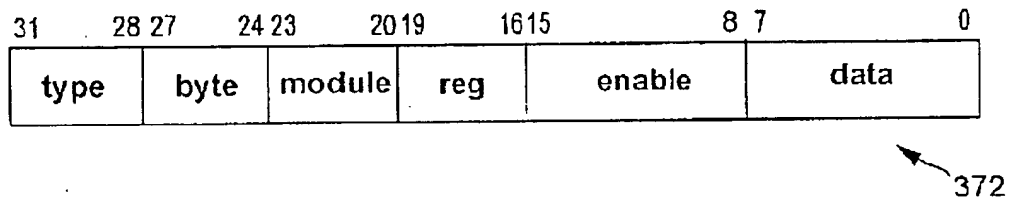


Fig. 21

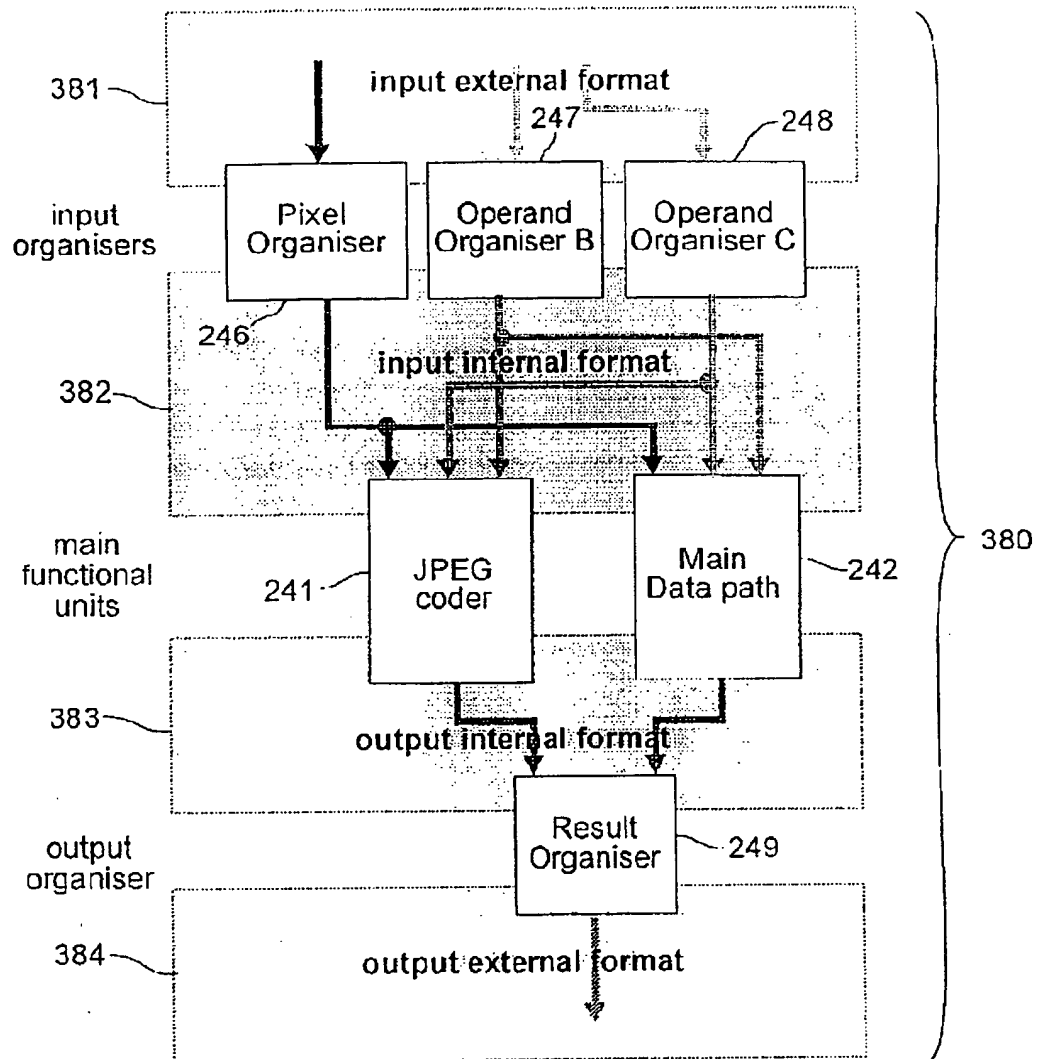


Fig. 22

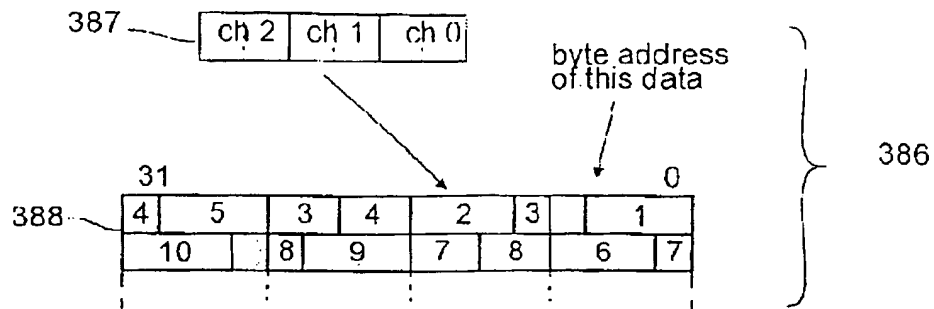


Fig. 23

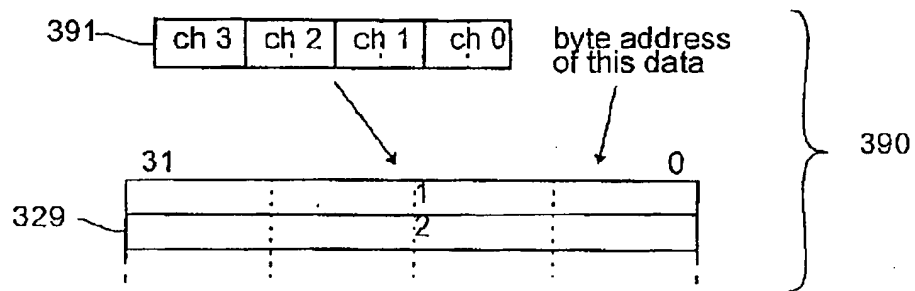


Fig. 24

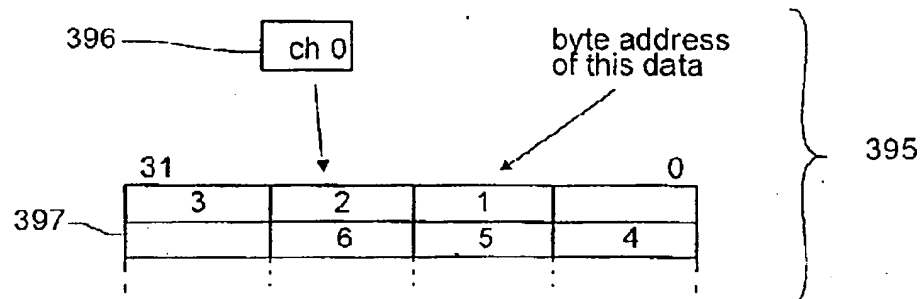


Fig. 25

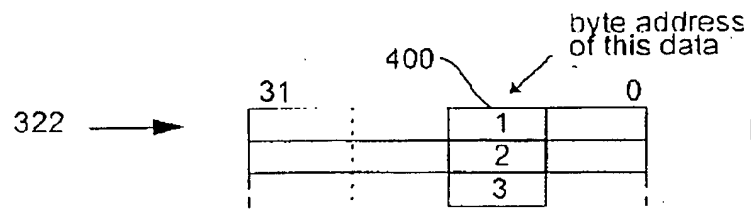


Fig. 26

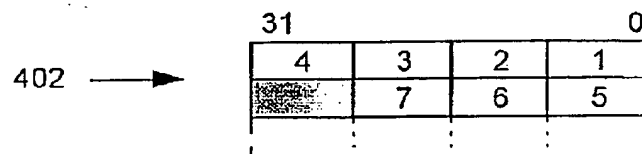


Fig. 27

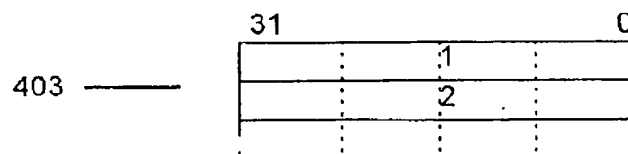


Fig. 28

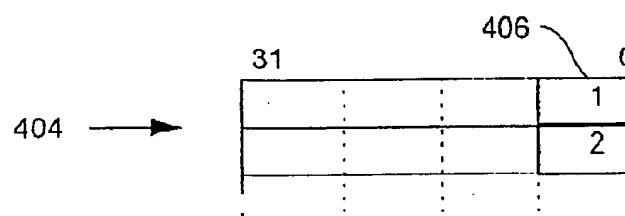


Fig. 29

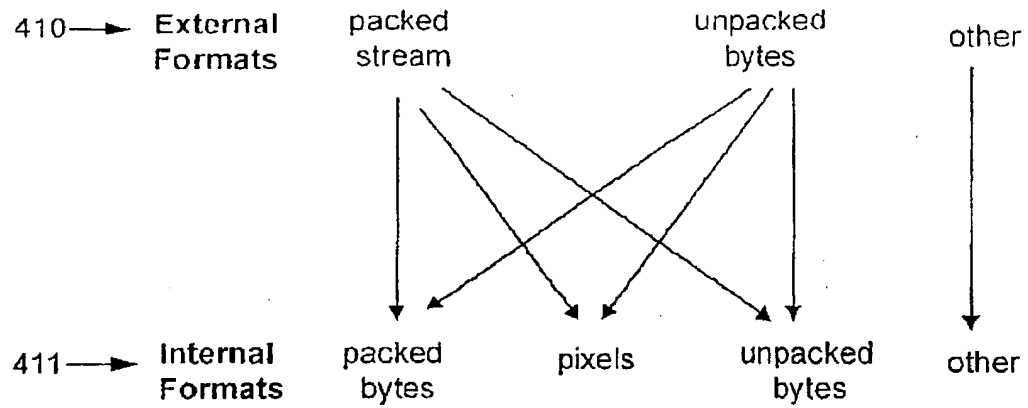


Fig. 30

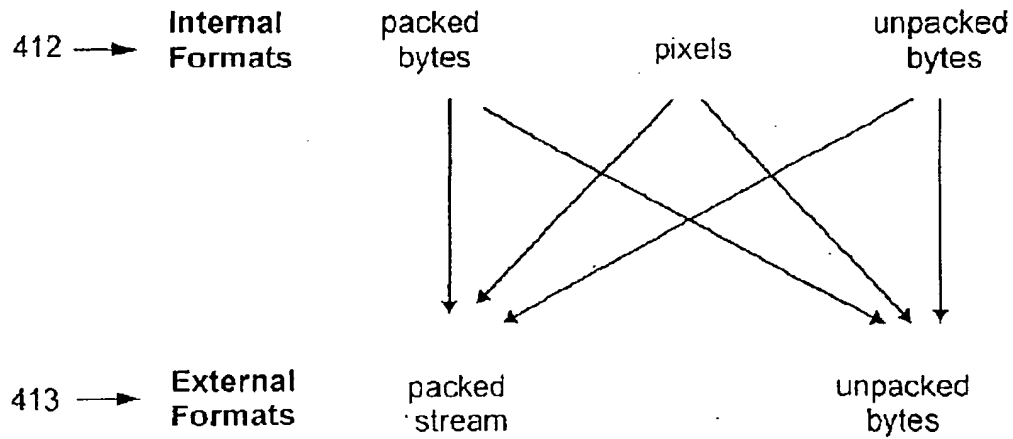


Fig. 31

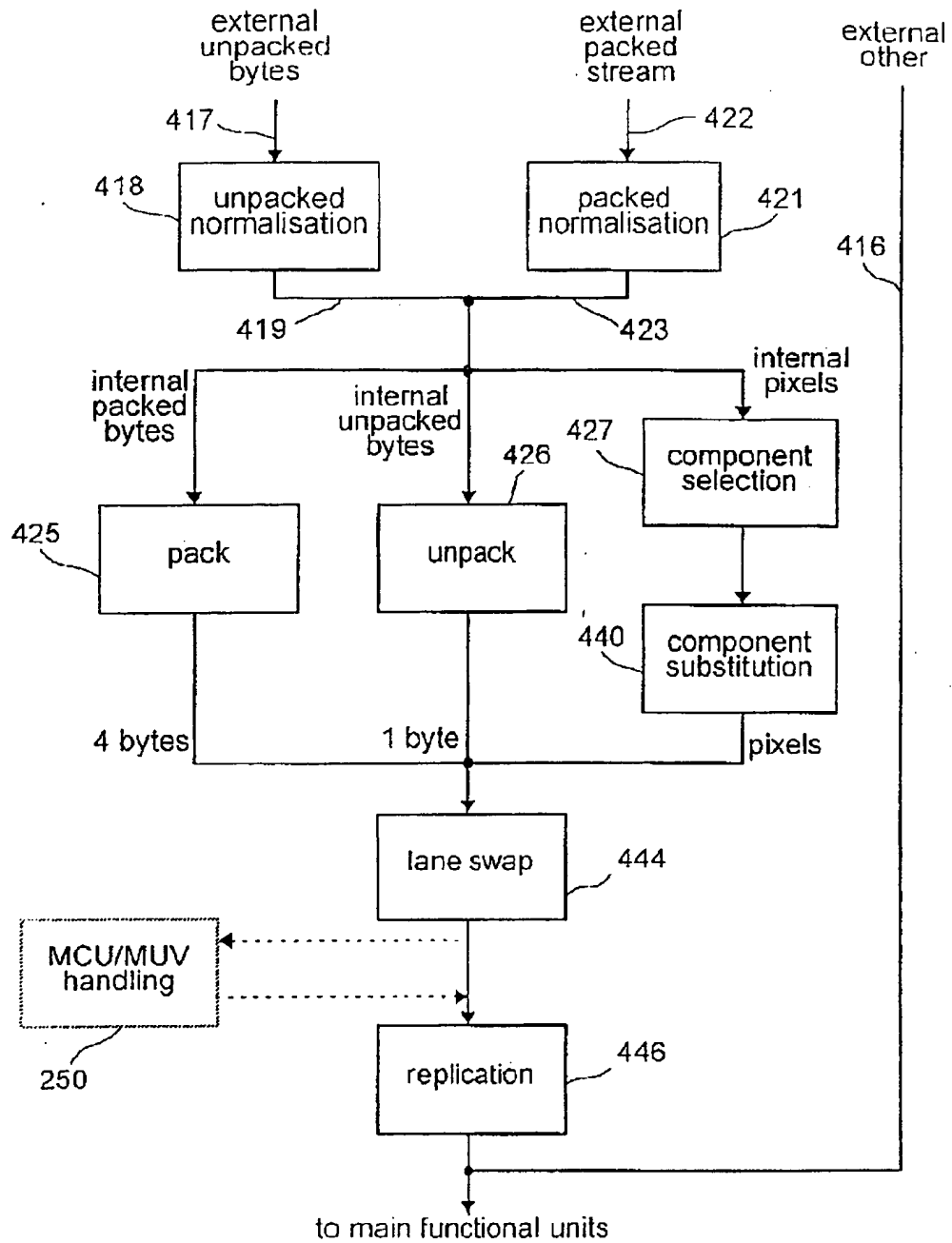
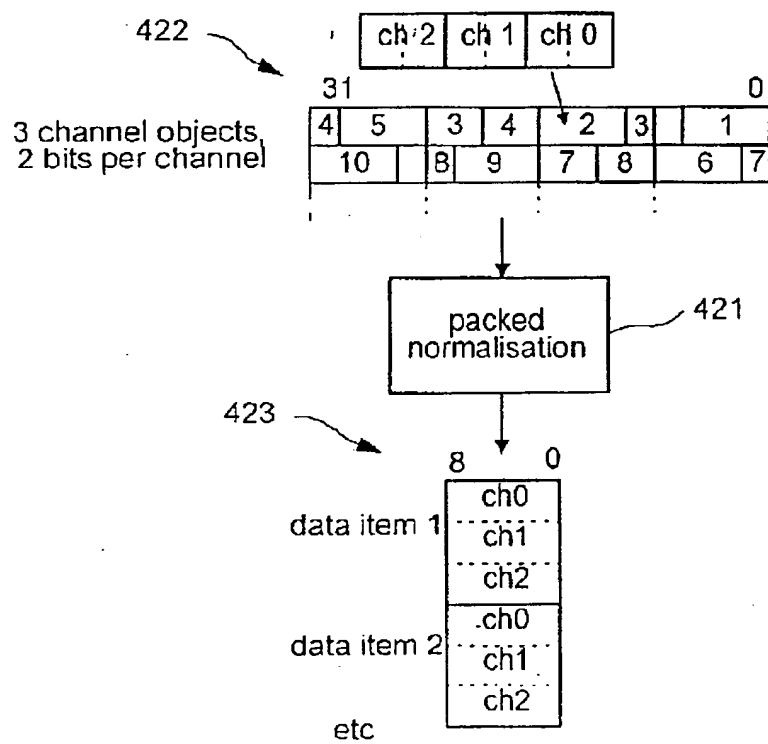
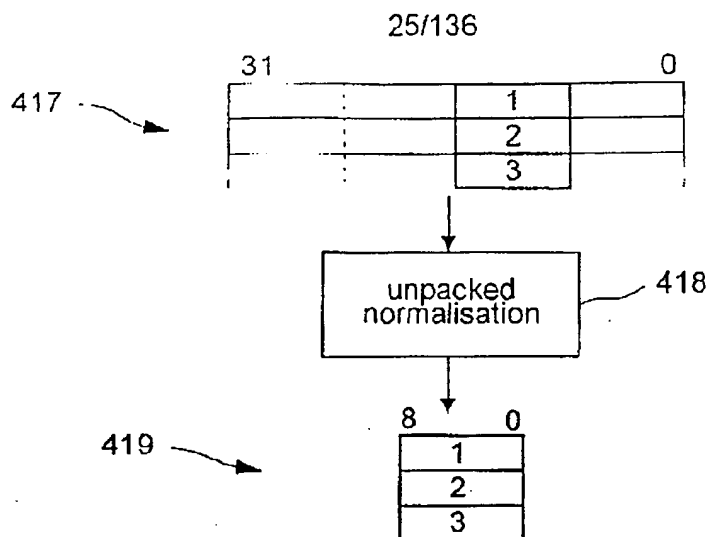


Fig. 32



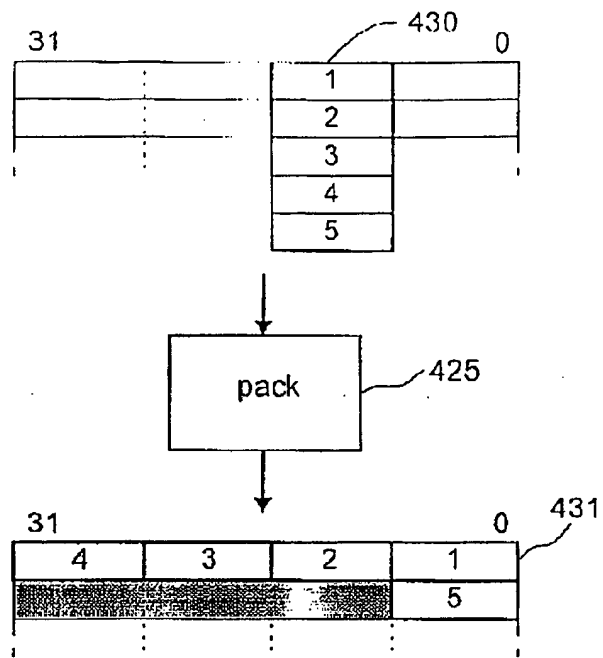


Fig. 35

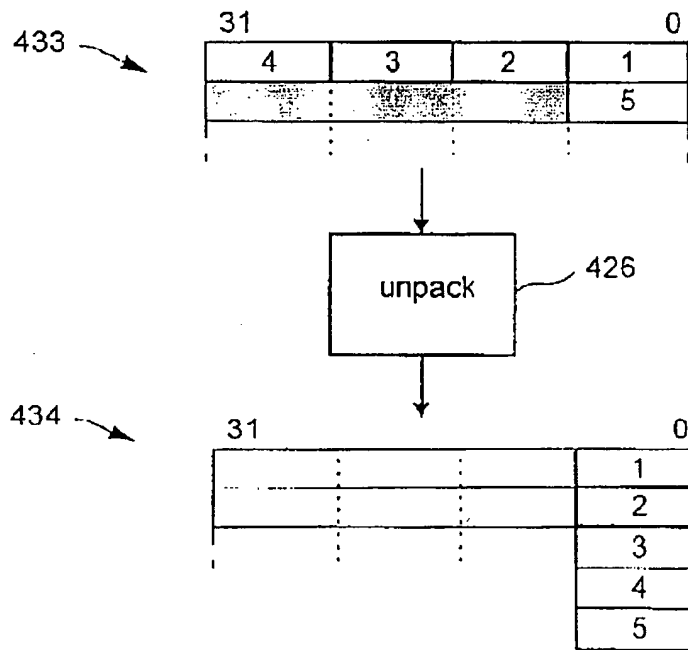


Fig. 36

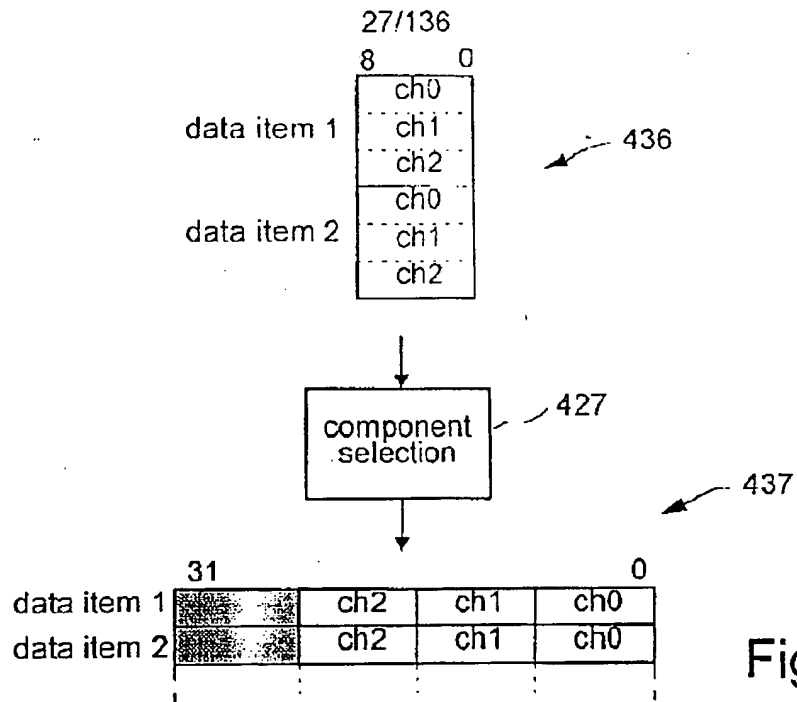


Fig. 37

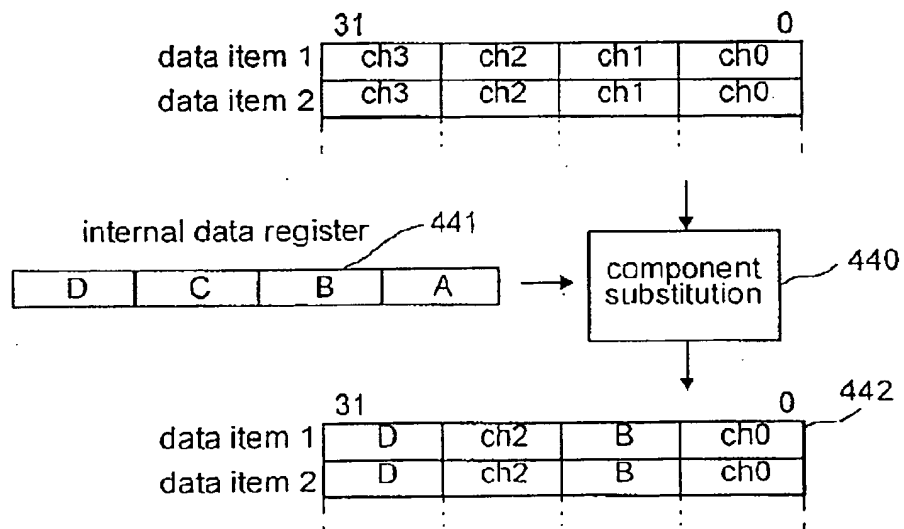


Fig. 38

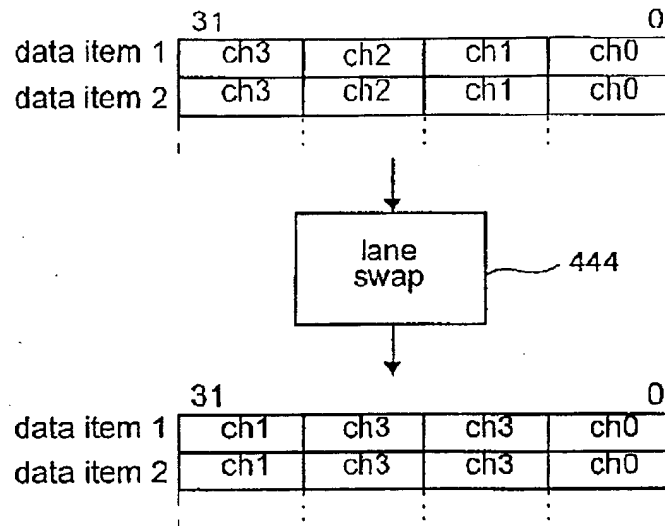


Fig. 39

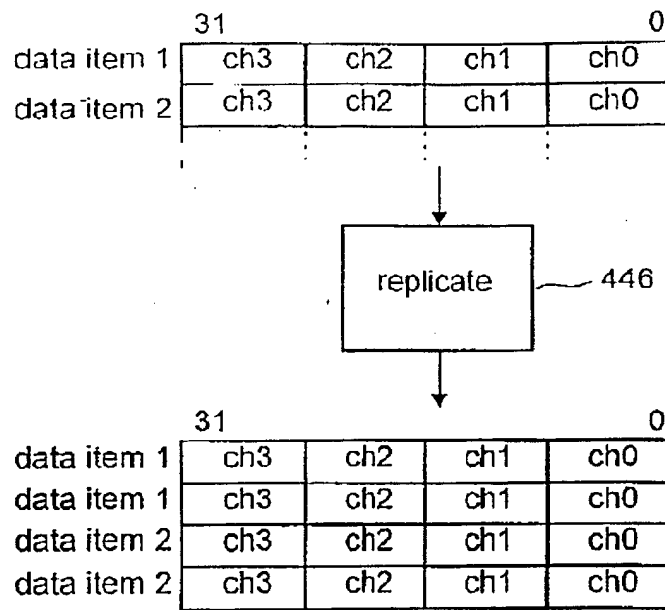


Fig. 40

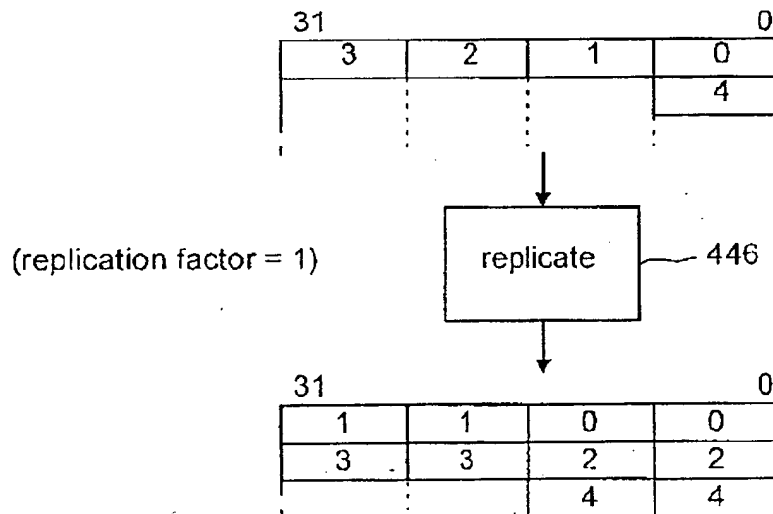


Fig. 41

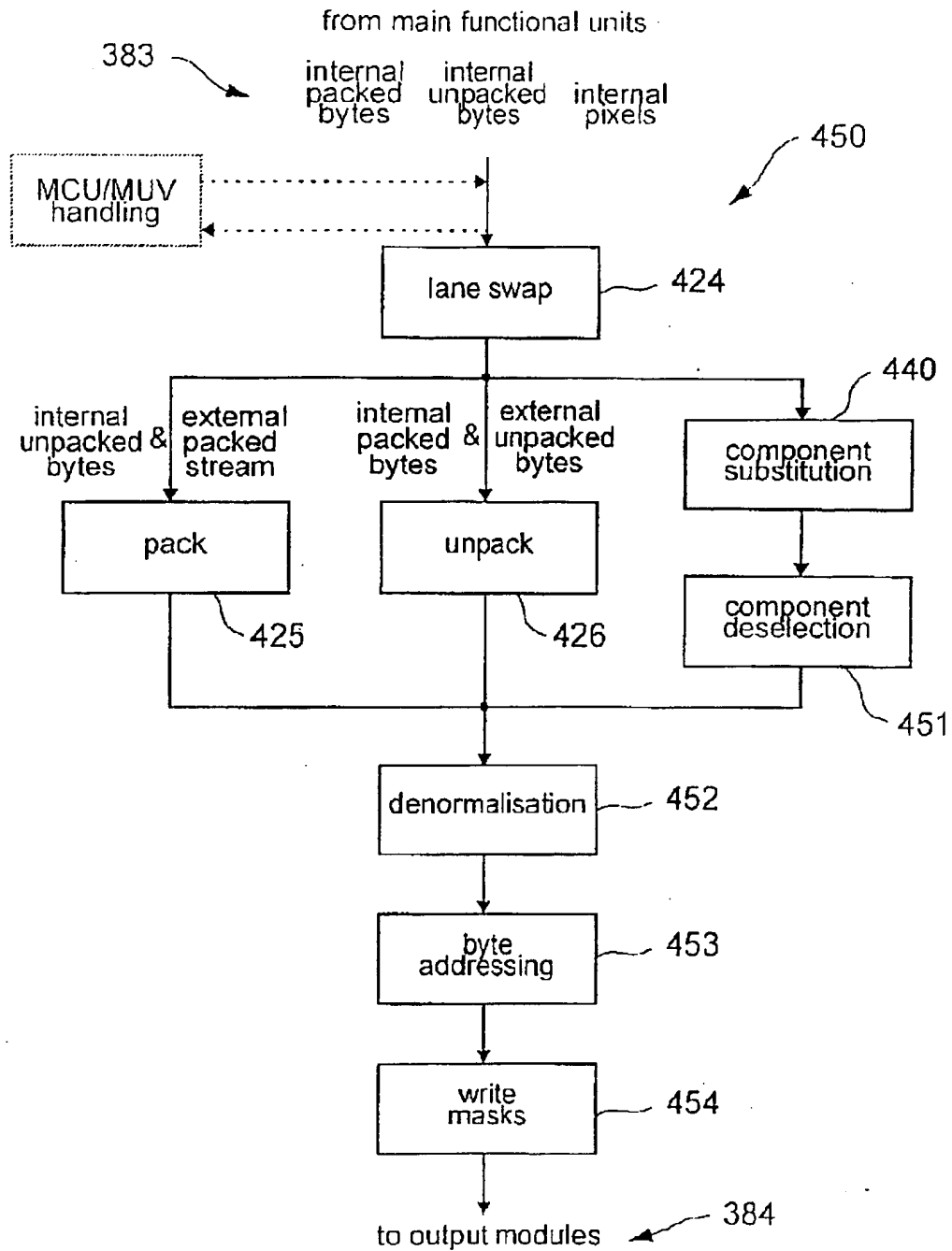


Fig. 42

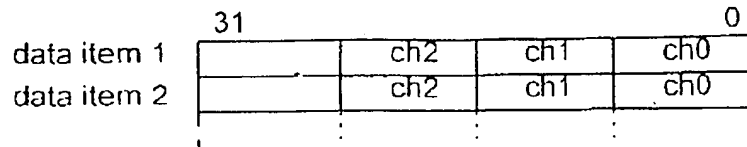


Fig. 43

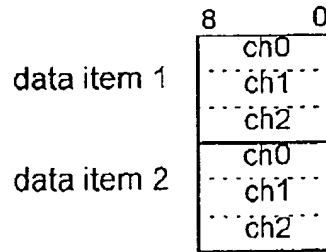
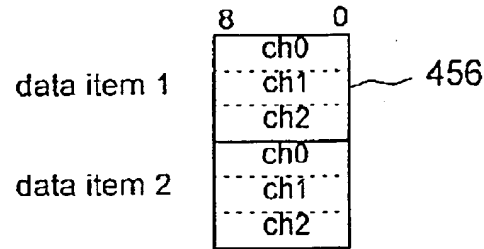
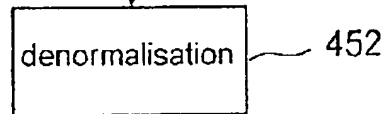
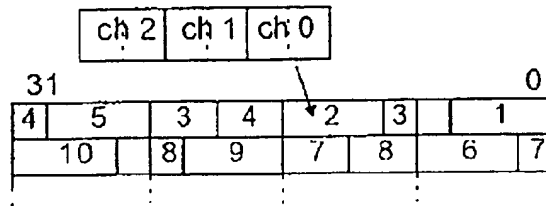


Fig. 44



3 channel objects,
2 bits per channel



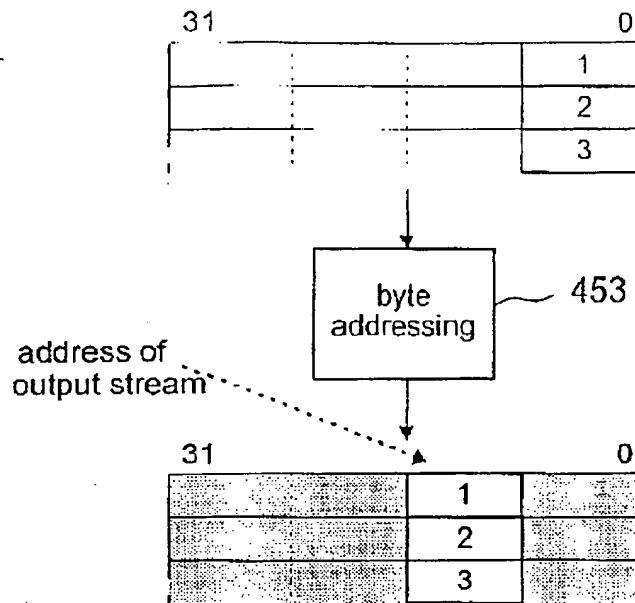


Fig. 45

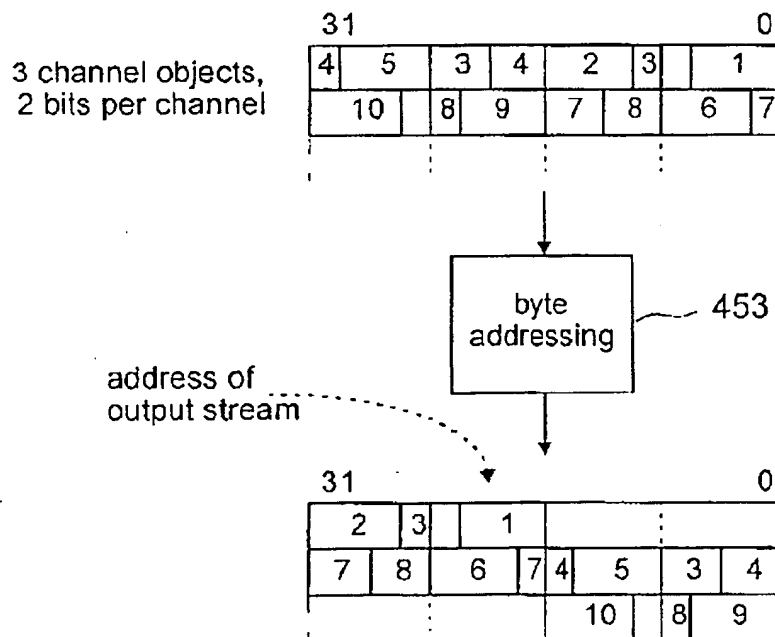


Fig. 46

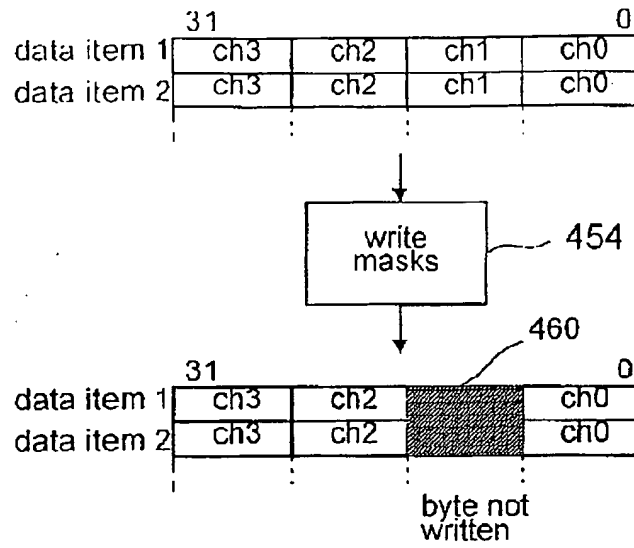
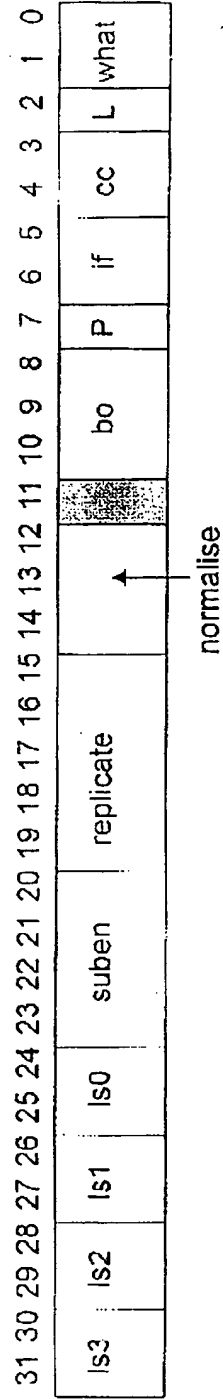


Fig. 47

Input Data Manipulation Registers (po_dmr, oob_dmr and ooc_dmr):



Output Data Manipulation Register (ro_dmr):

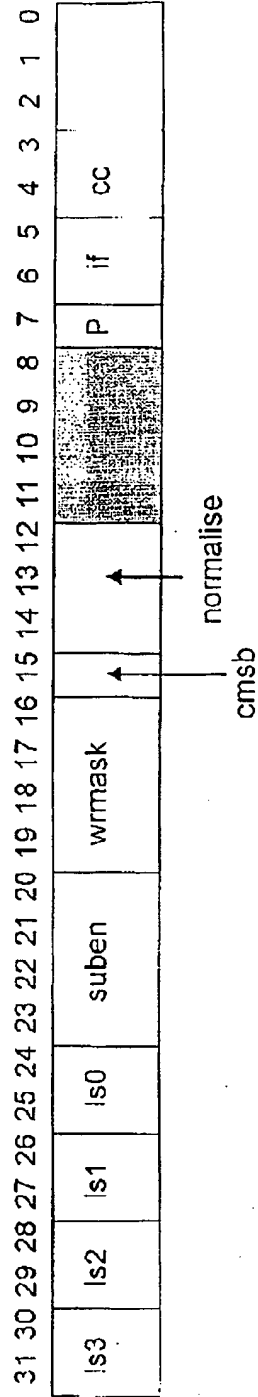


Fig. 48

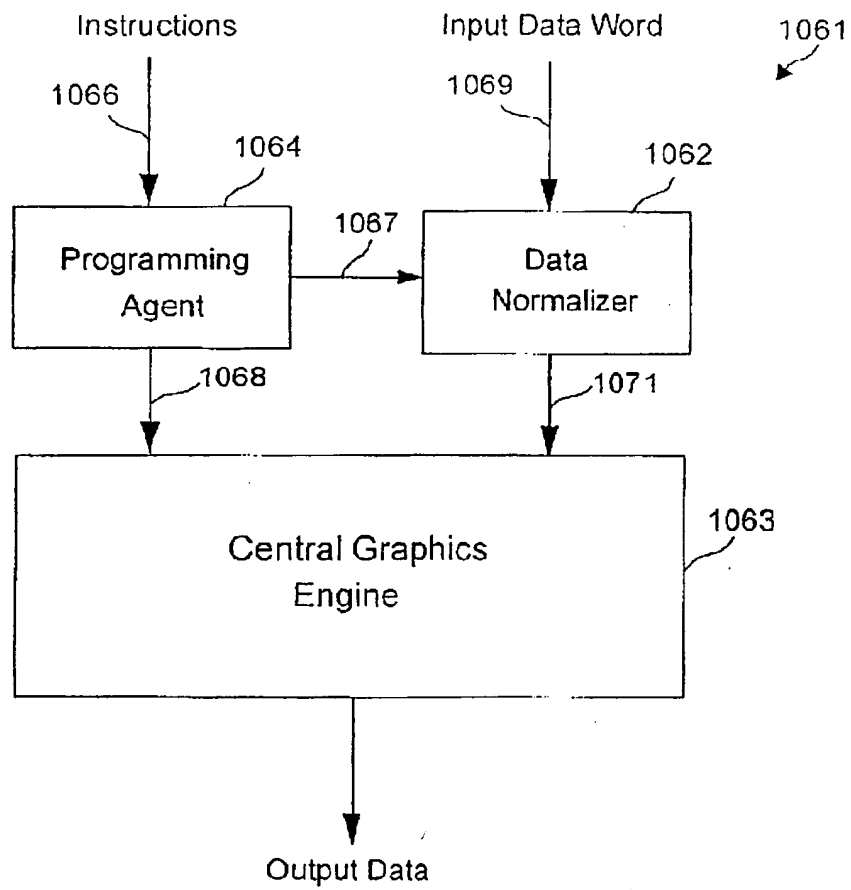


Fig. 49

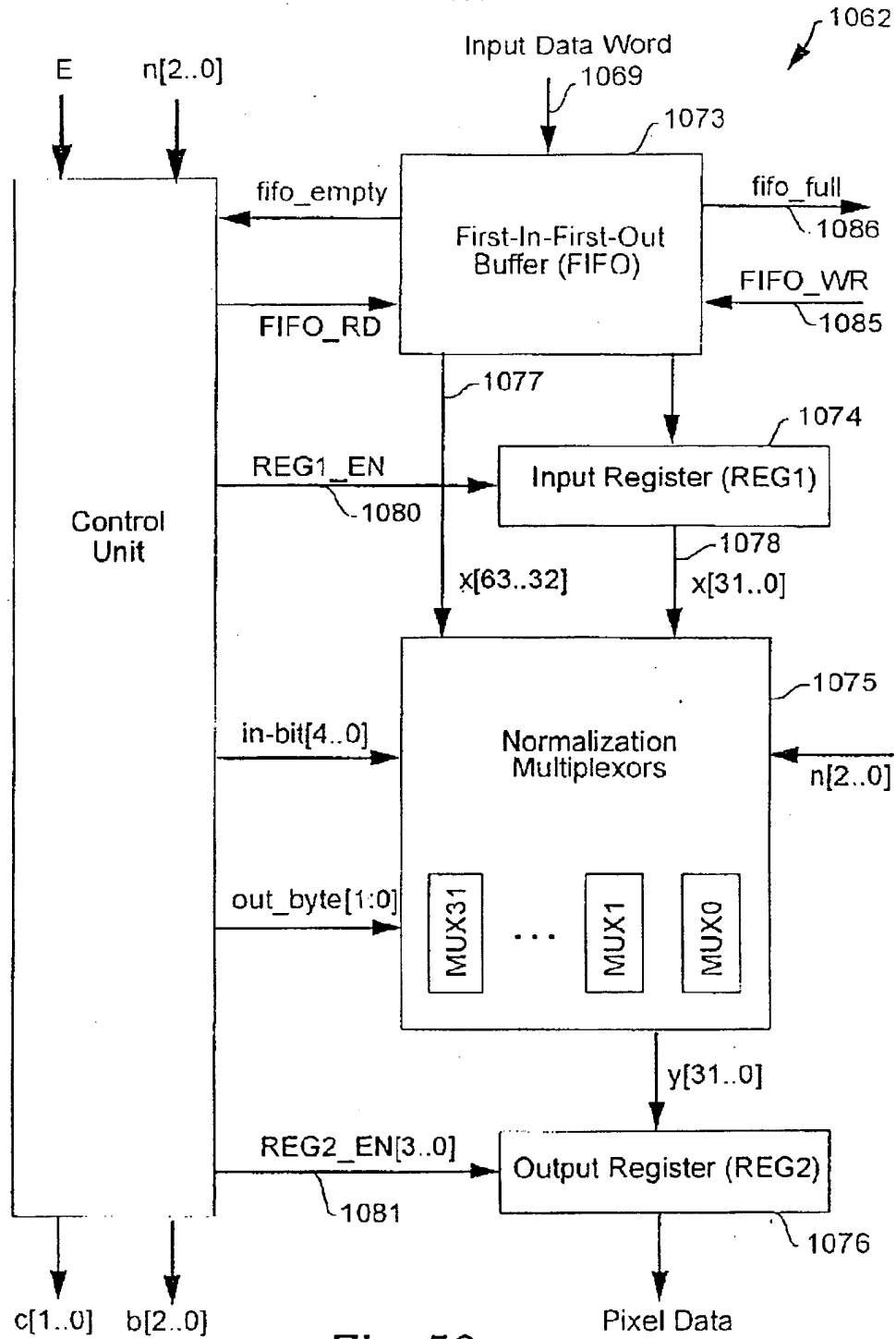


Fig. 50

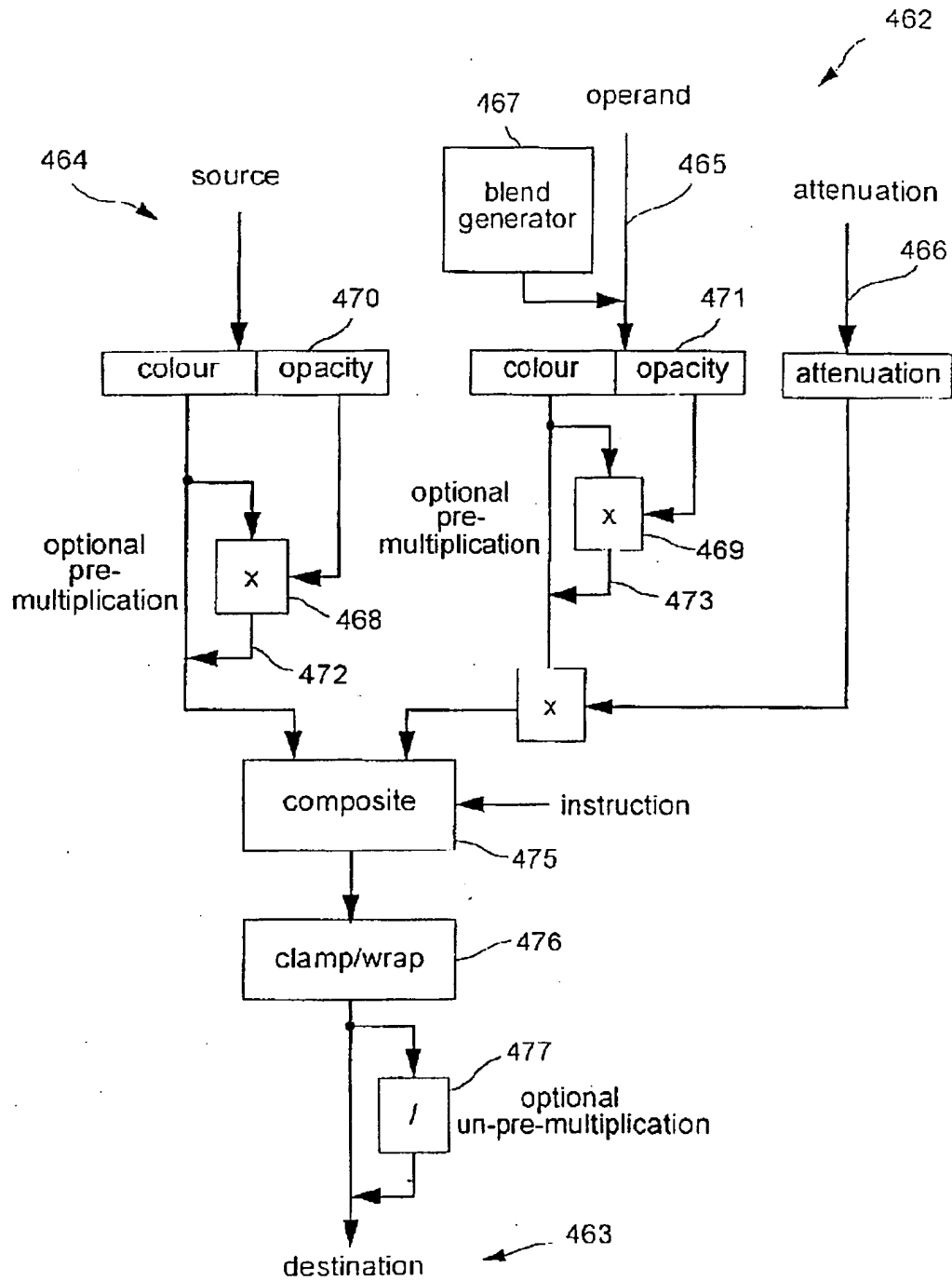


Fig. 51

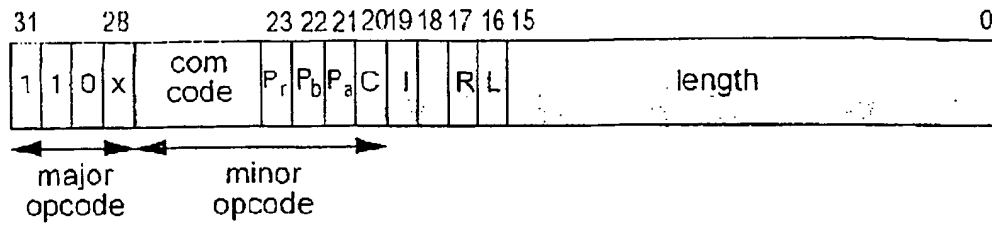


Fig. 52

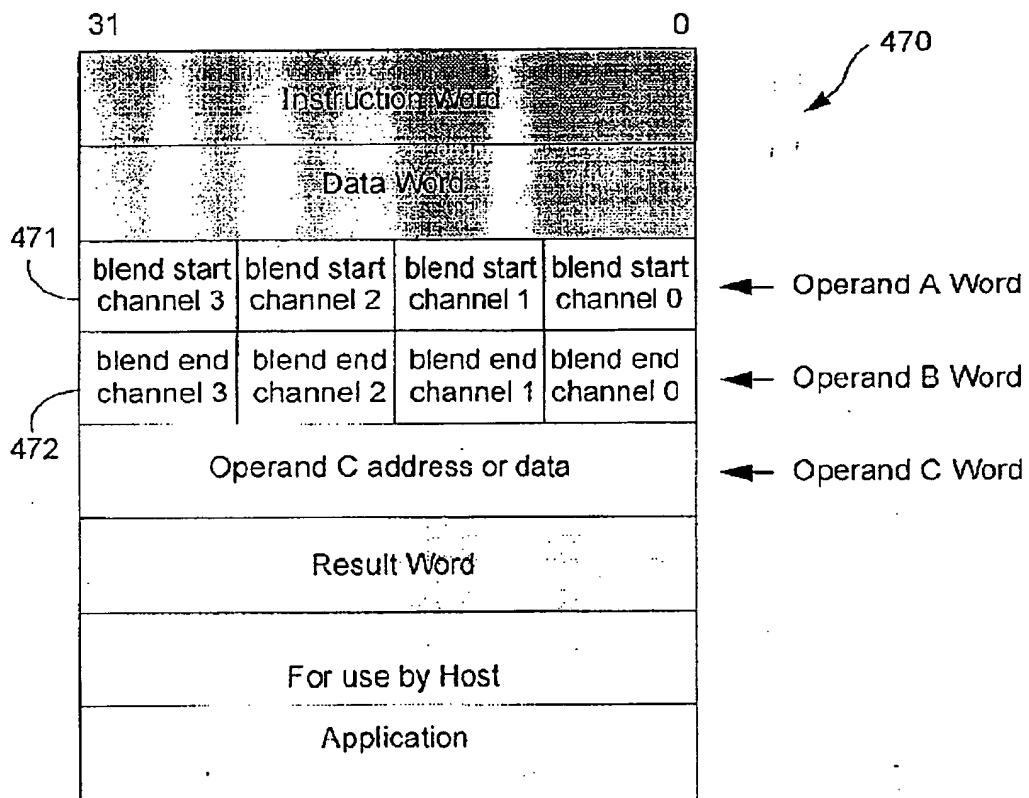


Fig. 53

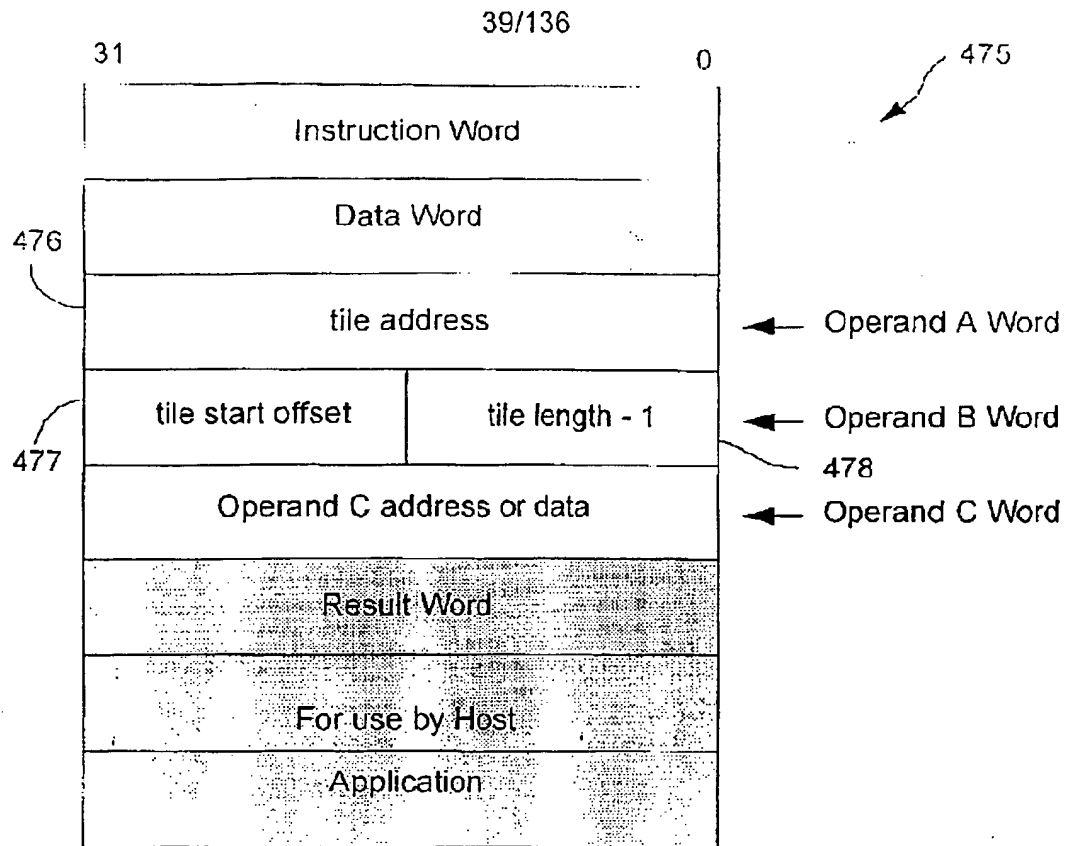


Fig. 54

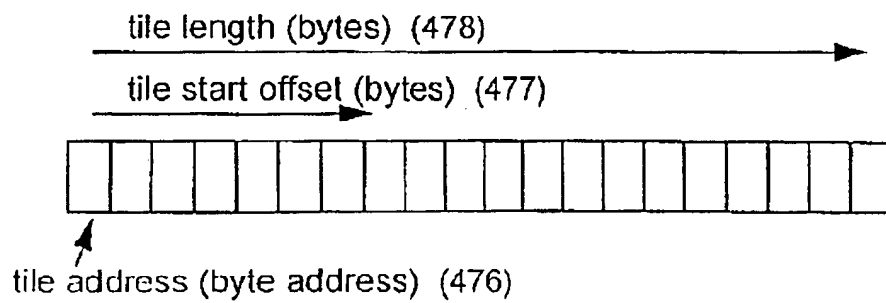


Fig. 55

Colour Value 0 0 0 0 0 1 0 0 481 480

Interval Table (4 bit)		Fractional Table (8 bit)	
482		483	
0	0	0	/256
1	0	1	38 /256
2	0	2	80 /256
3	0	3	125 /256
4	0	4	160 /256
5	0	5	202 /256
6	0	6	240 /256
7	1	7	0 /256
8	1	8	45 /256
9	1	9	85 /256
10	1	10	140 /256
11	1	11	200 /256
485		486	
249	13	249	/256
250	14	250	0 /256
251	14	251	42 /256
252	14	252	81 /256
253	14	253	105 /256
254	14	254	180 /256
255	14	255	220 /256

Fig. 56

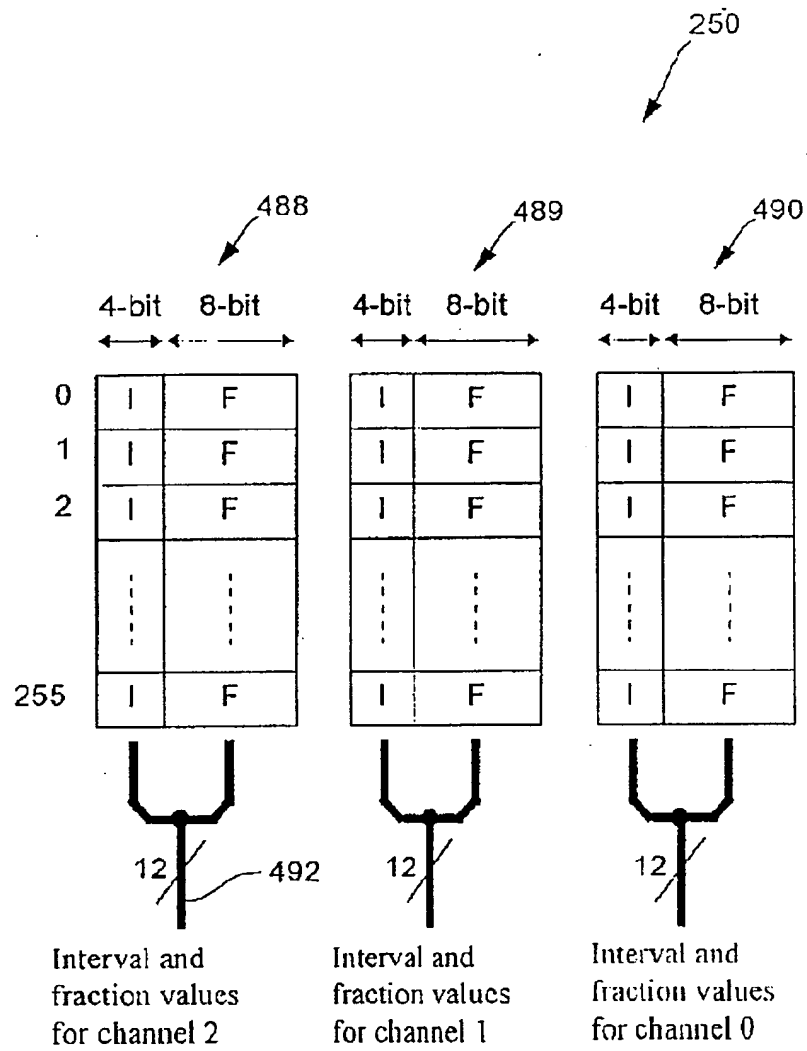
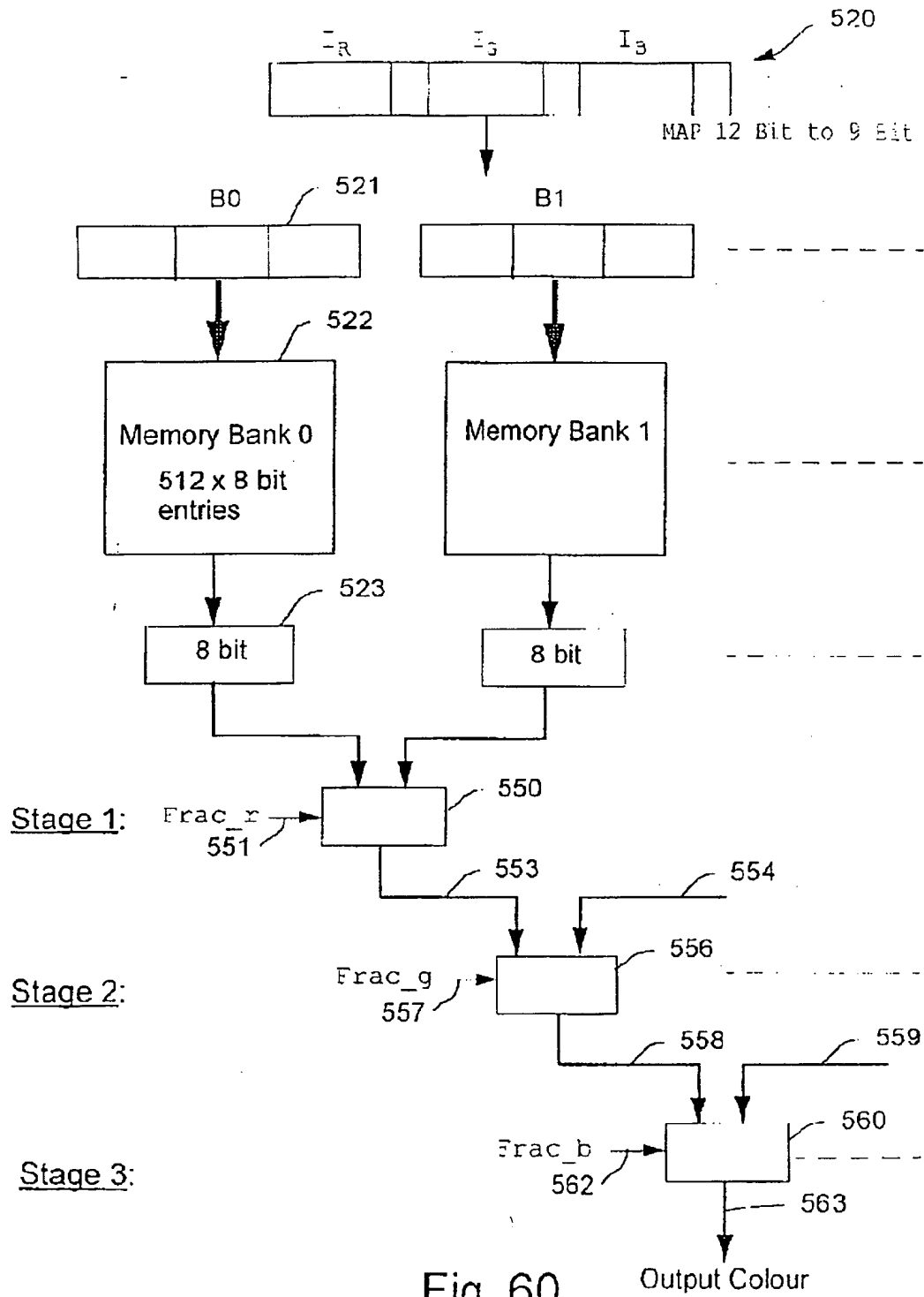


Fig. 57



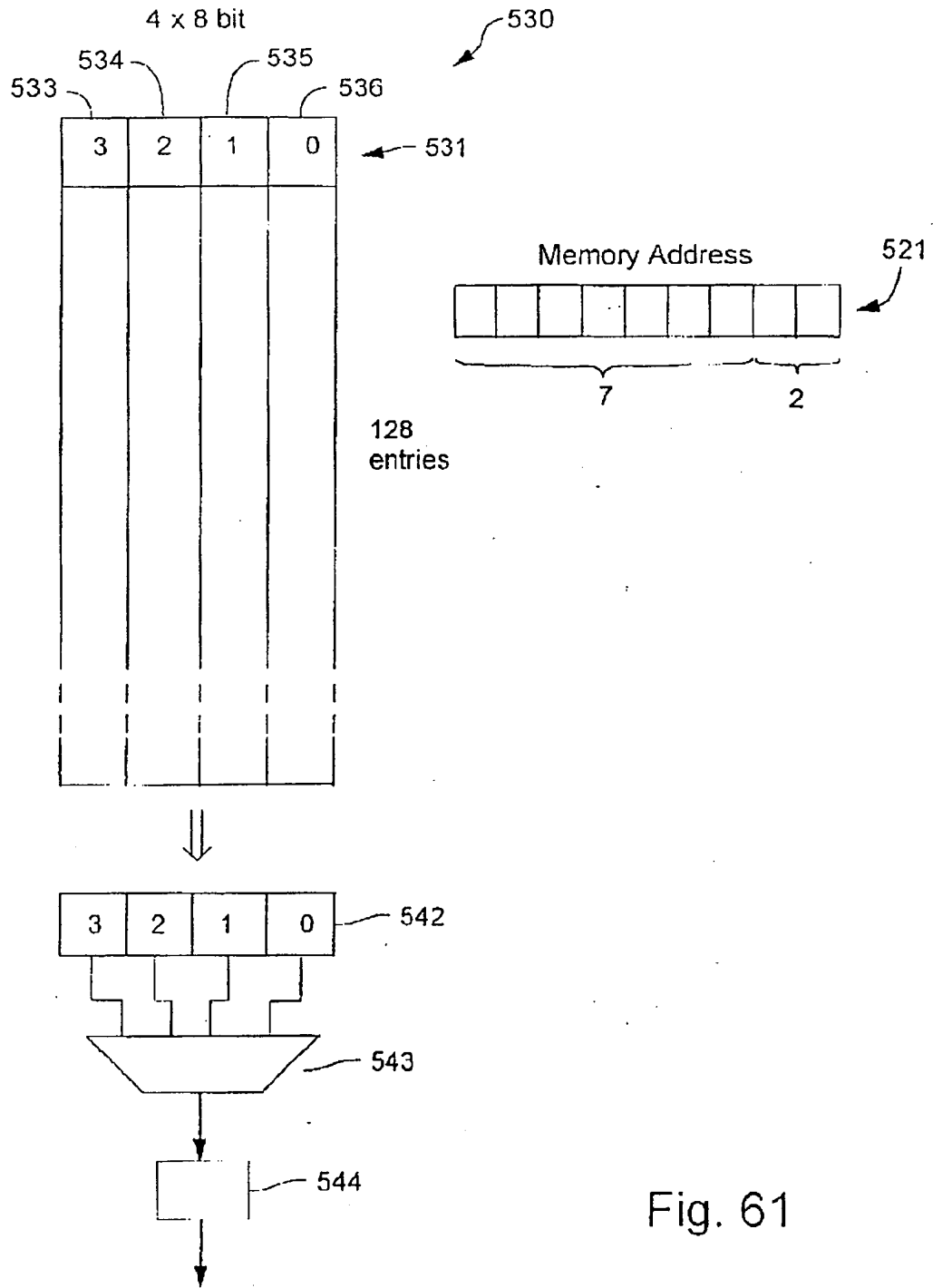


Fig. 61

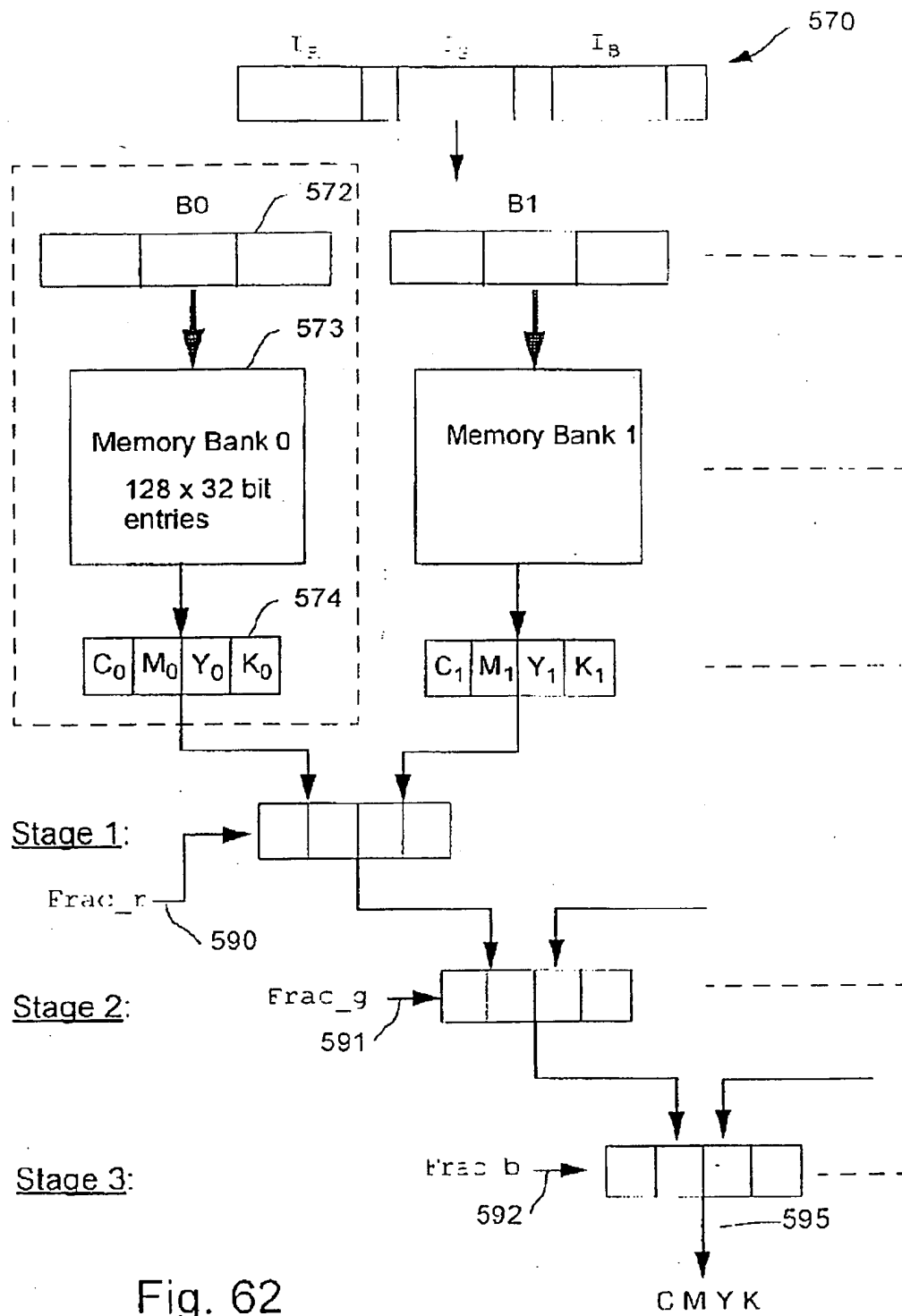


Fig. 62

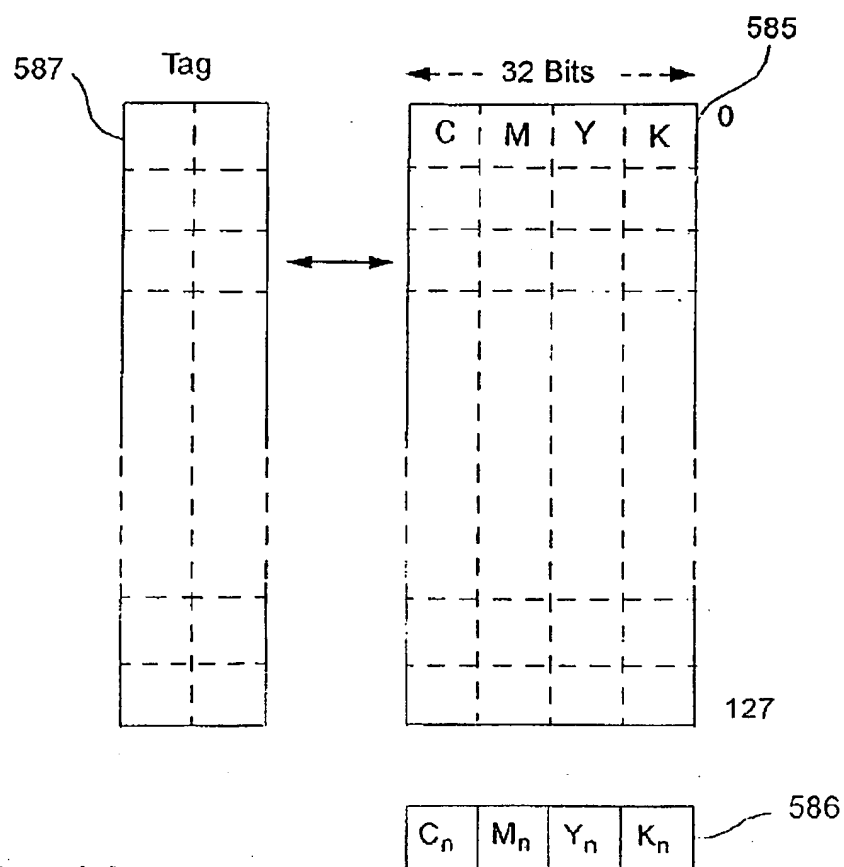
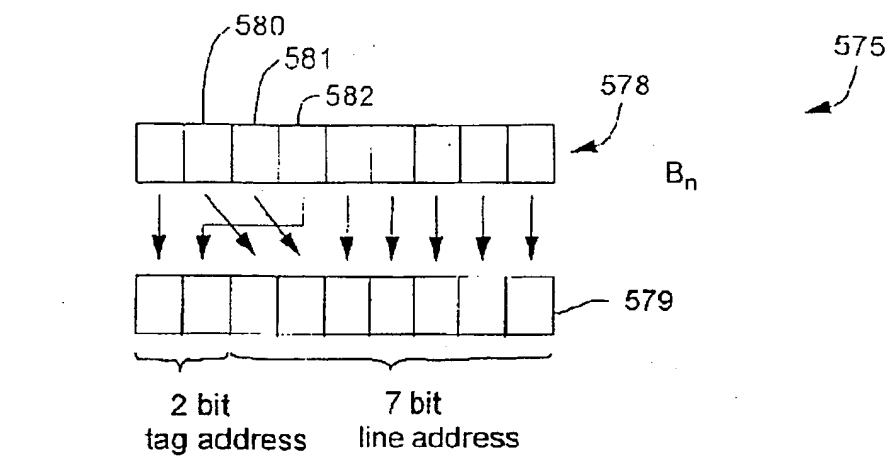


Fig. 63

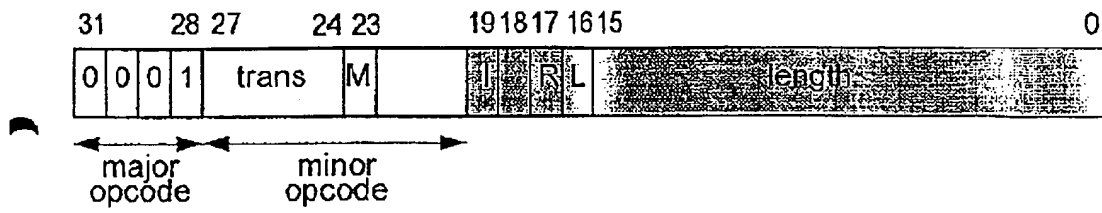
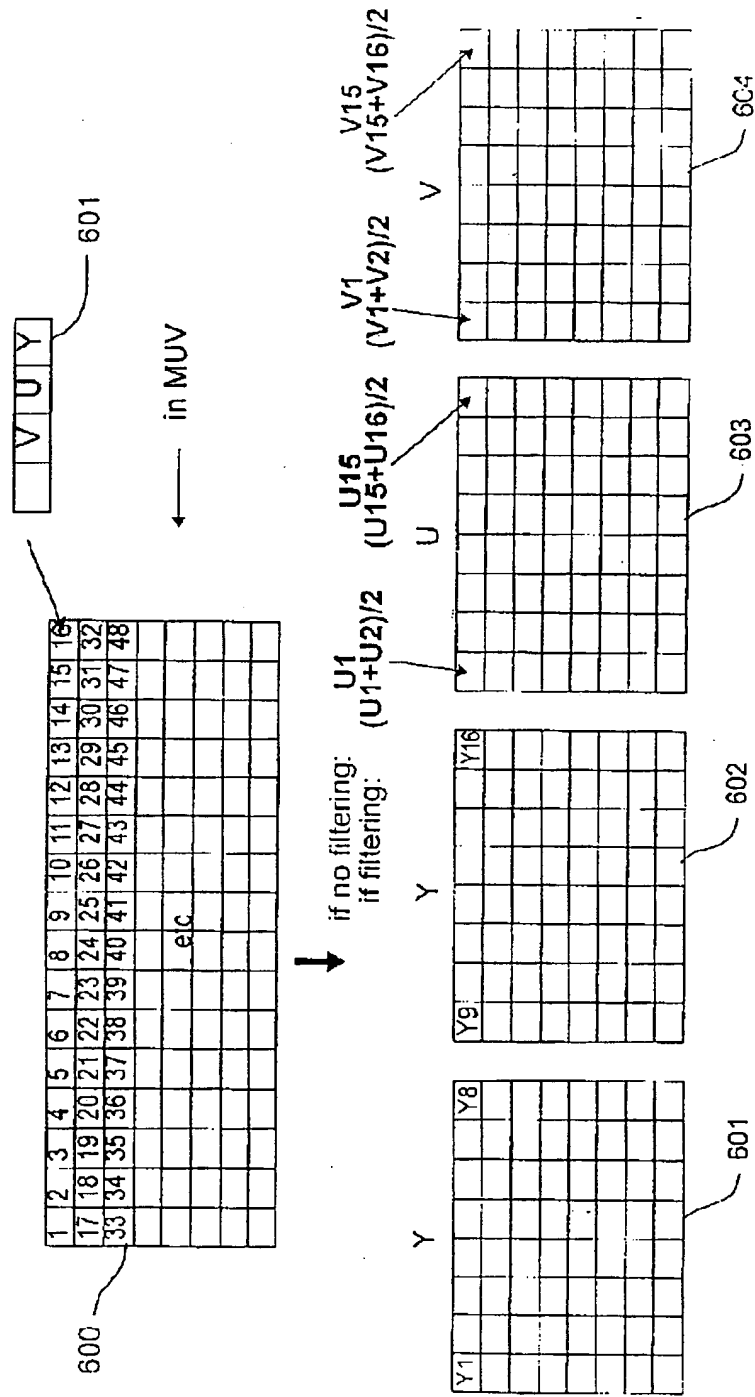


Fig. 64



Fig. 66



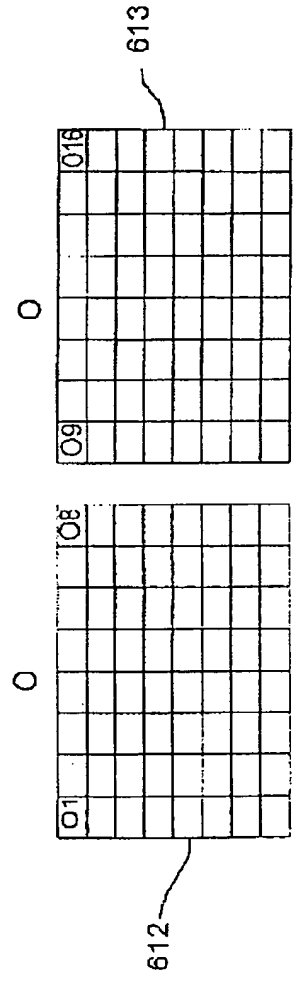
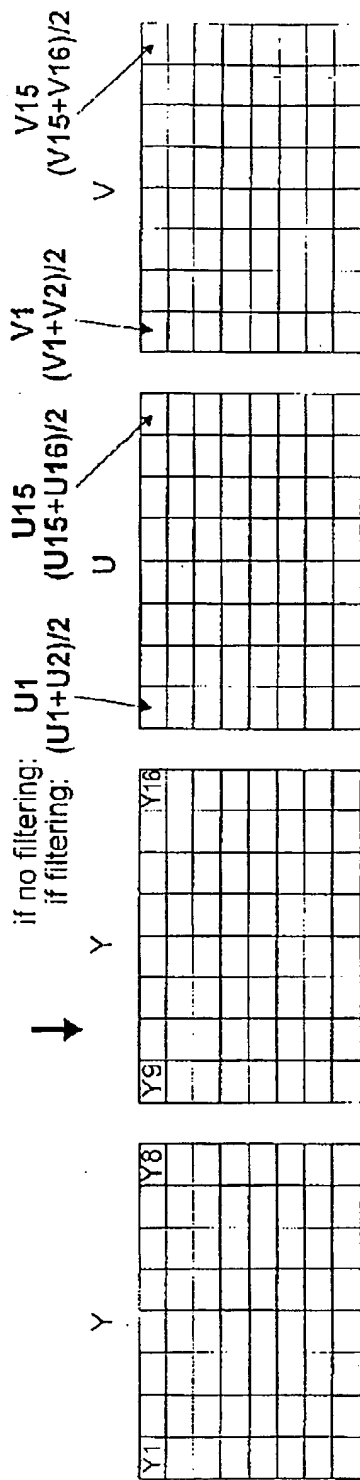
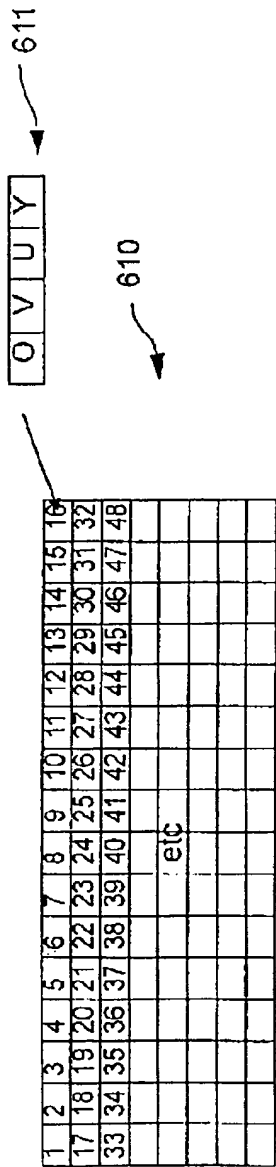


Fig. 67

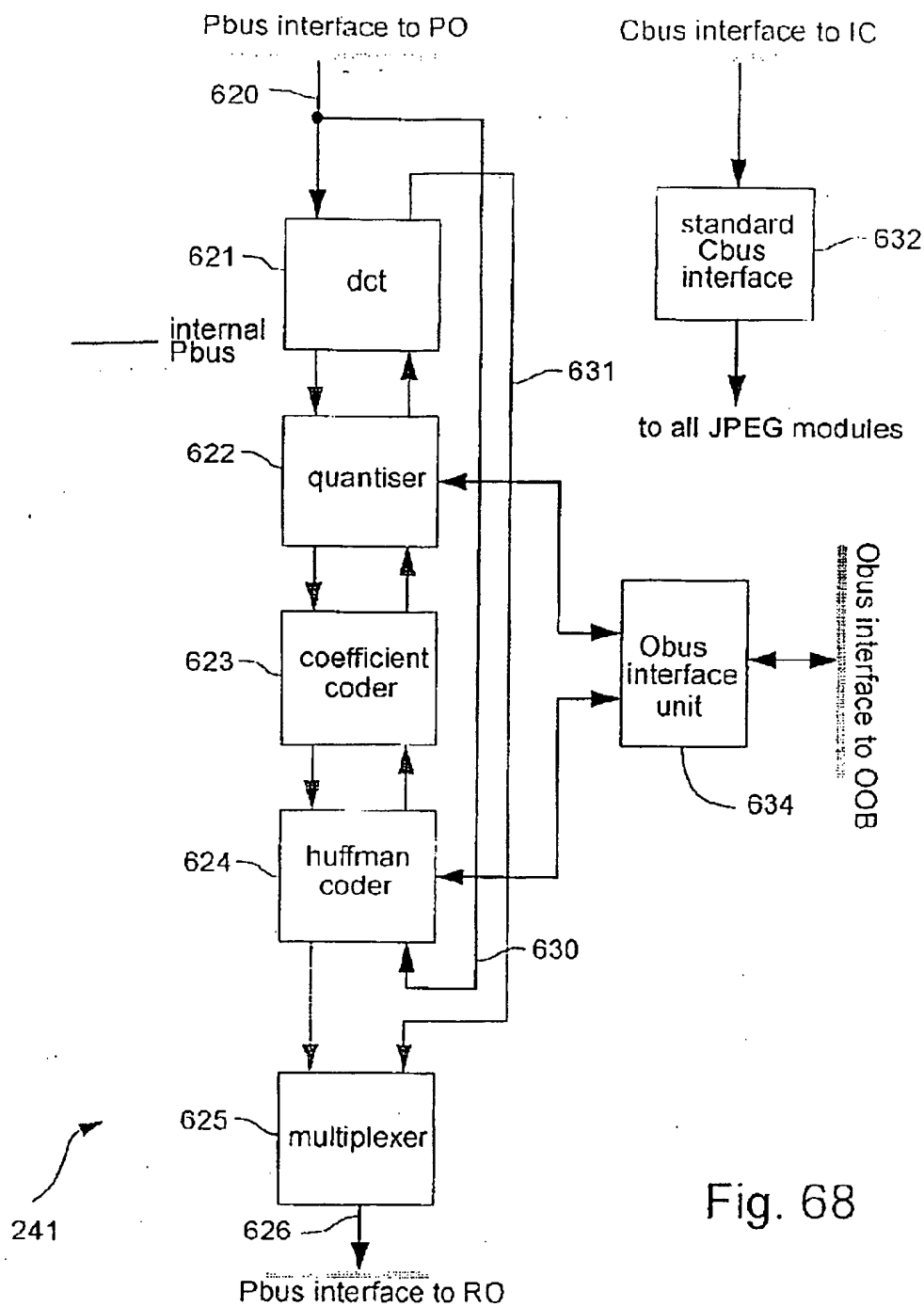


Fig. 68

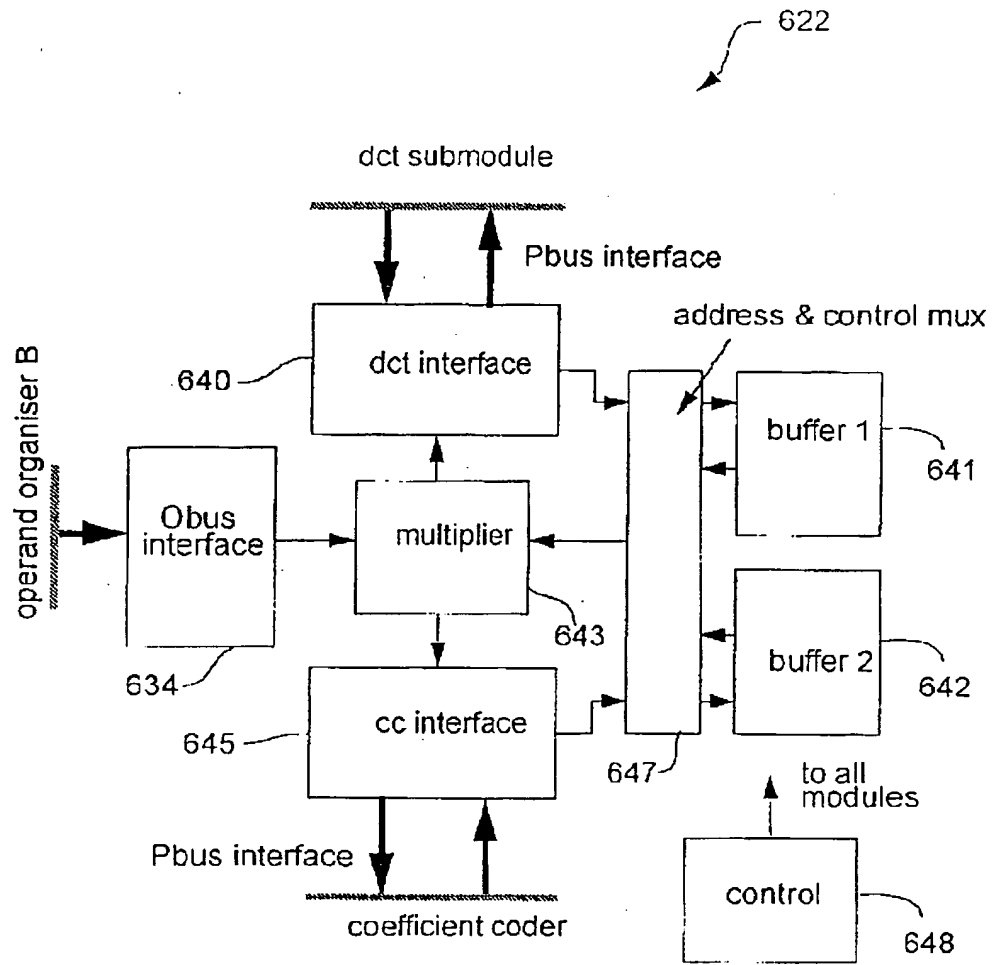


Fig. 69

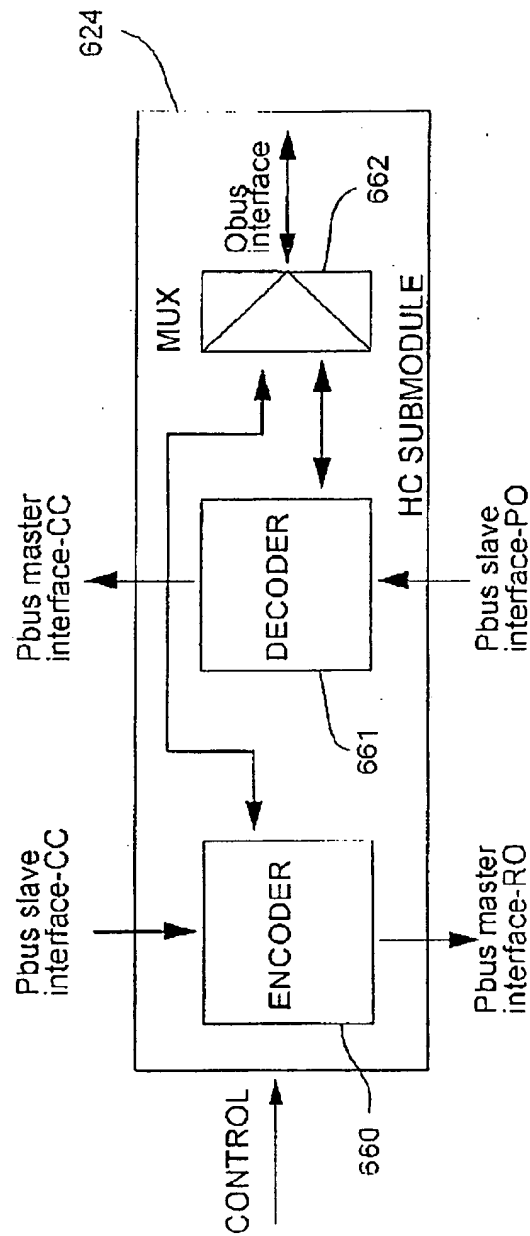


Fig. 70

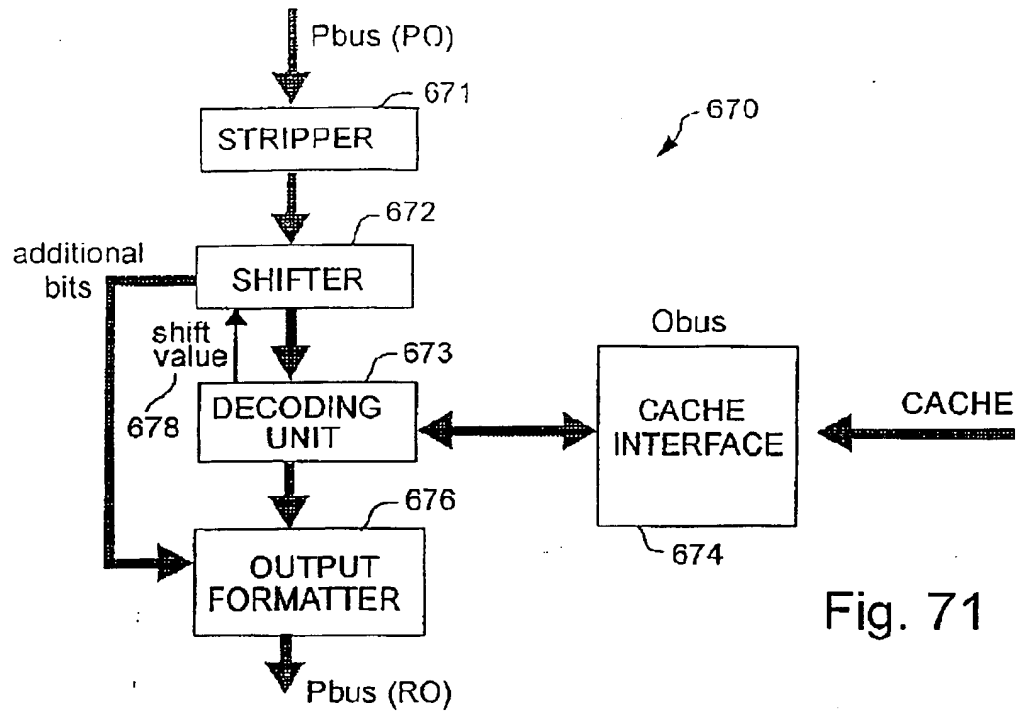


Fig. 71

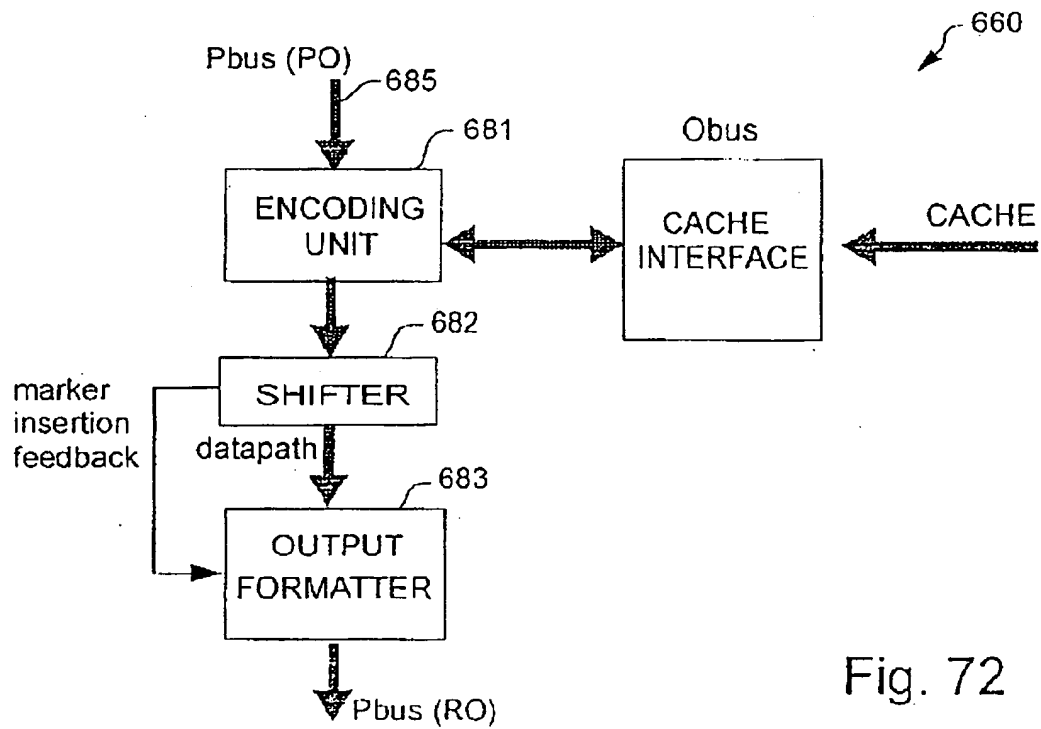


Fig. 72

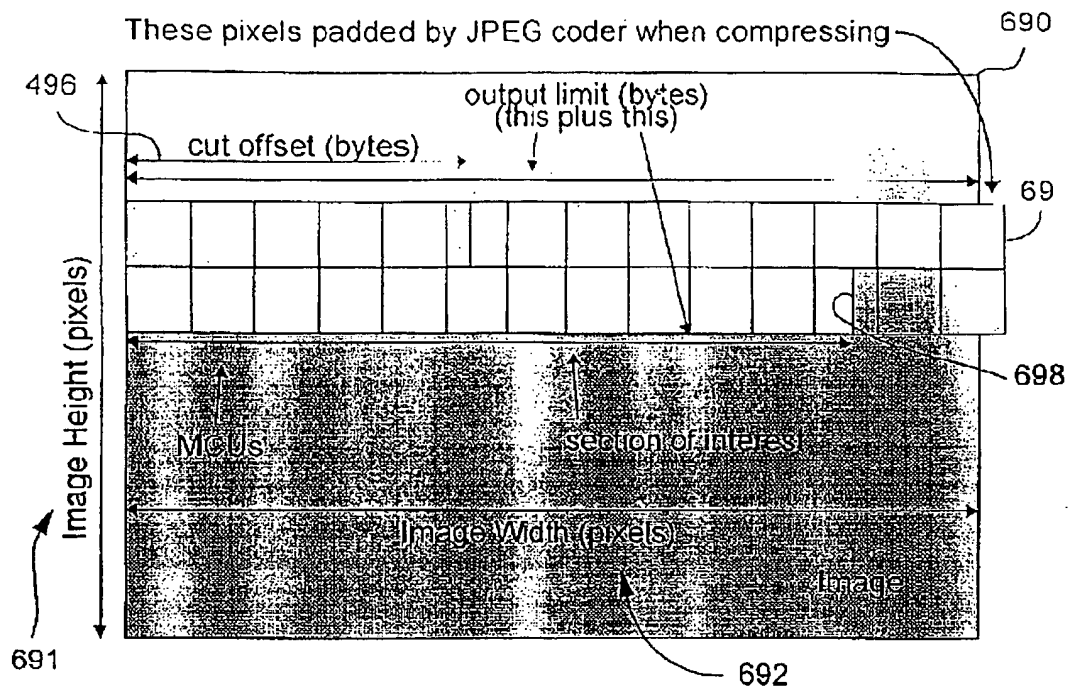


Fig. 73

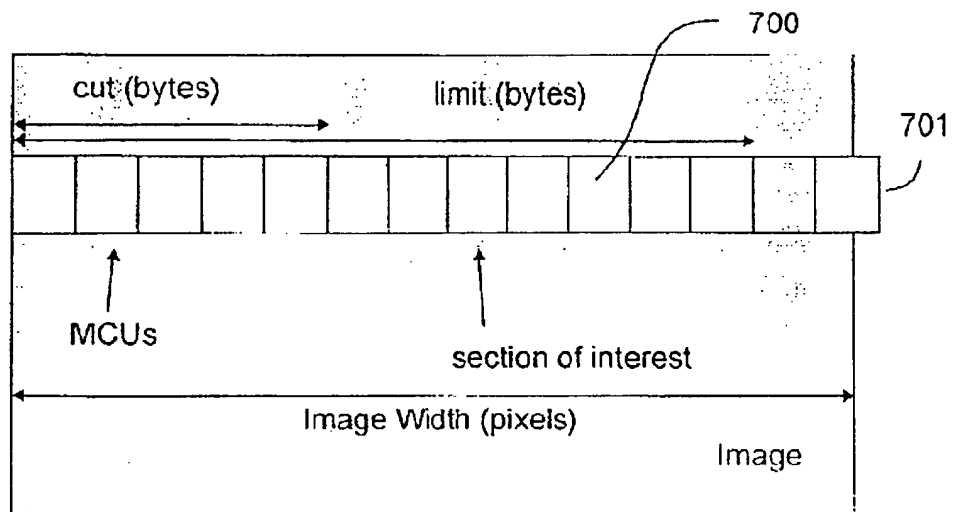


Fig. 74

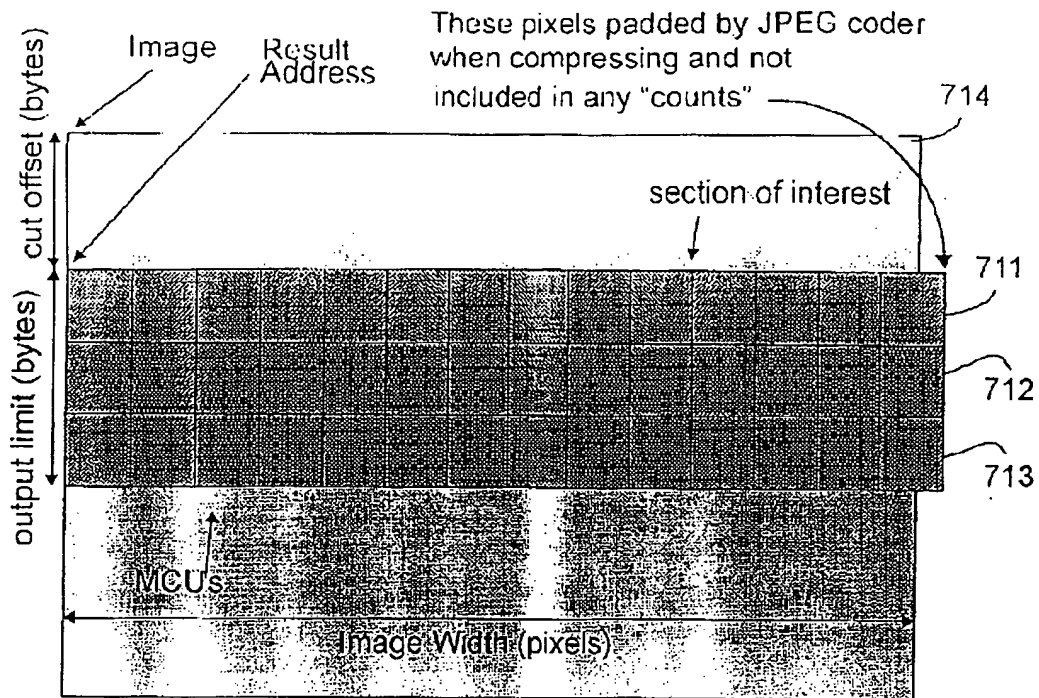


Fig. 75

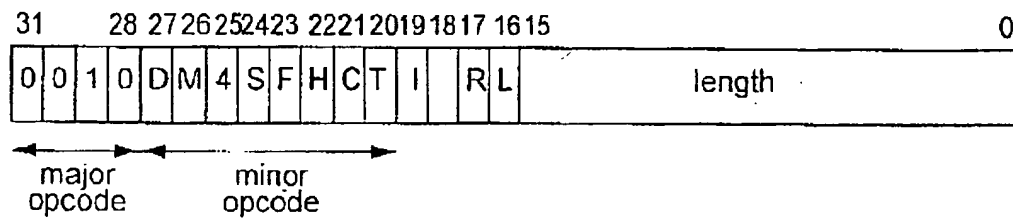


Fig. 76

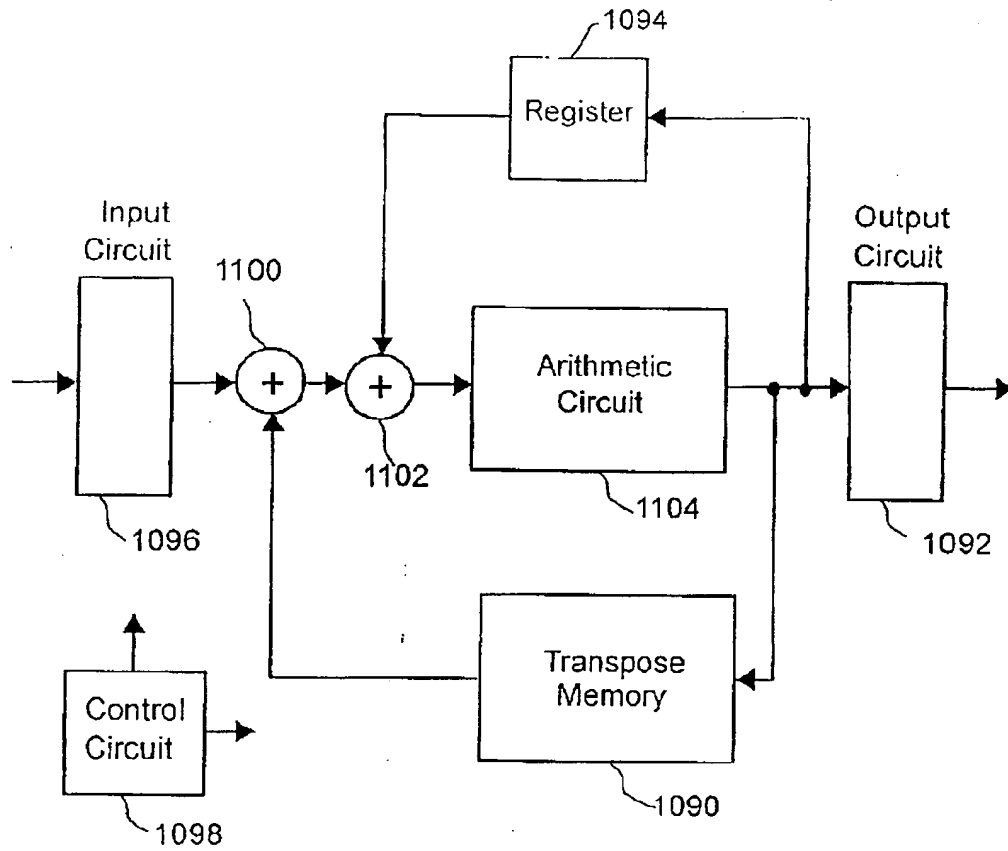


Fig. 77
(prior art)

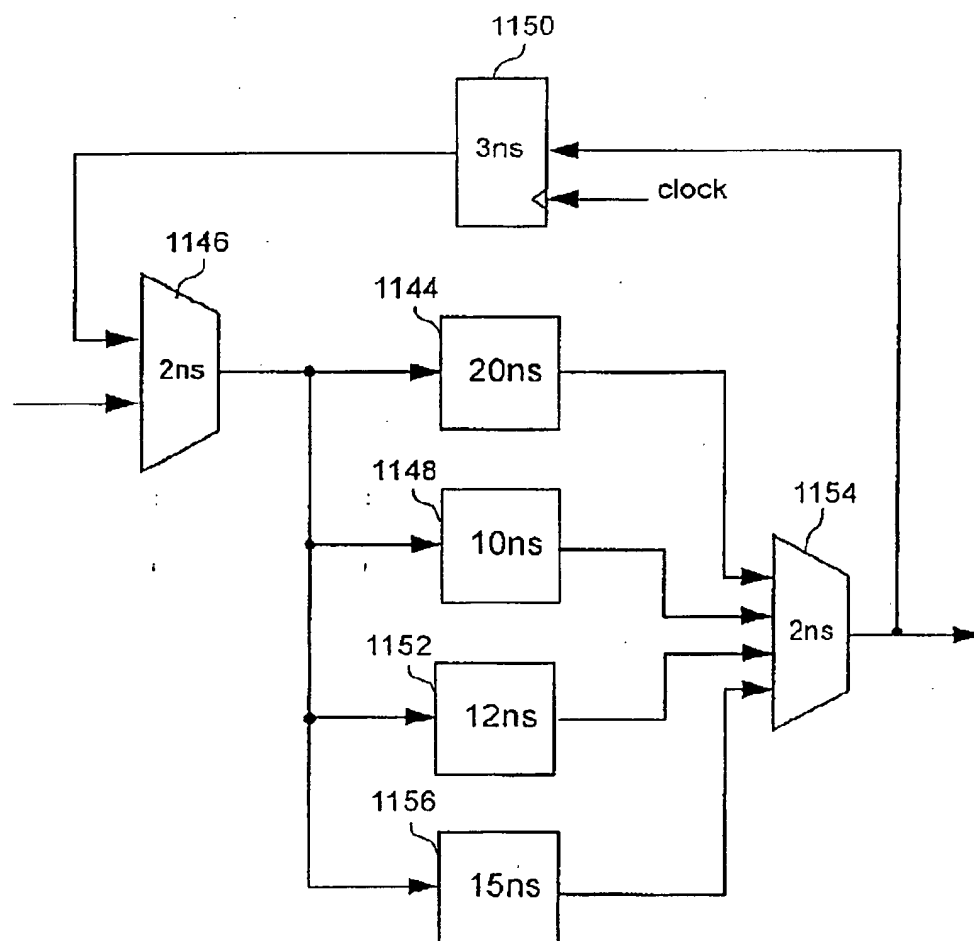


Fig. 78
(prior art)

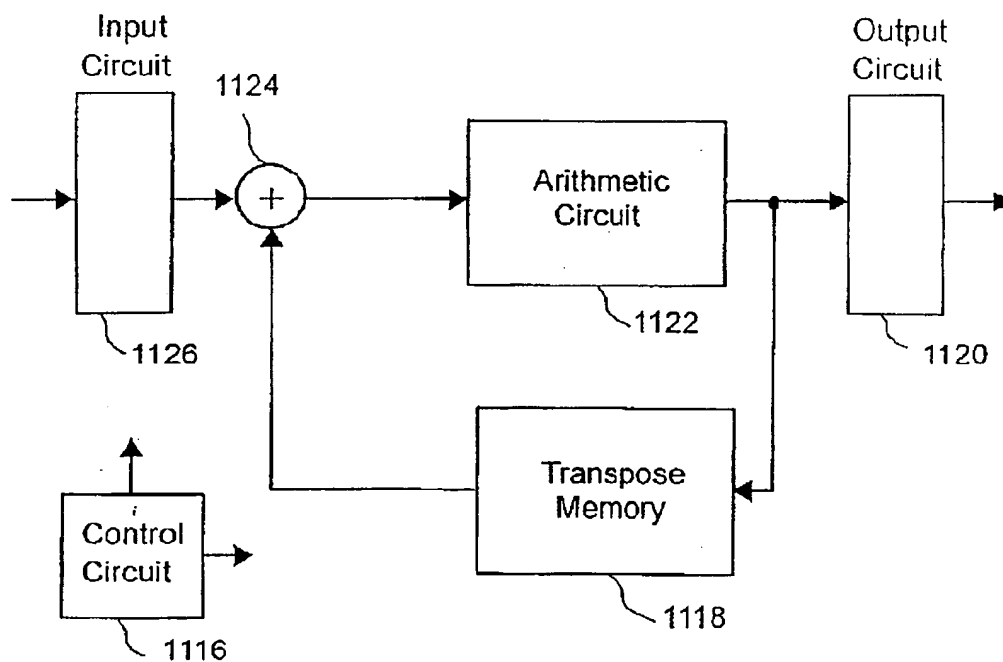


Fig. 79

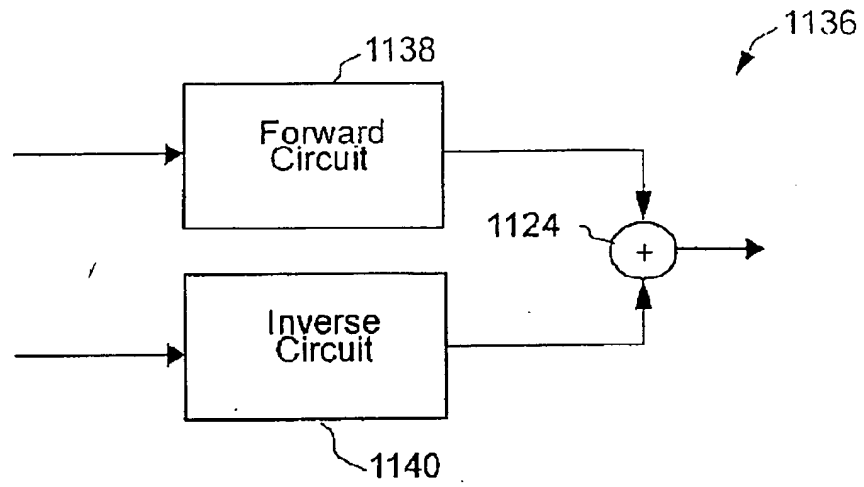


Fig. 80

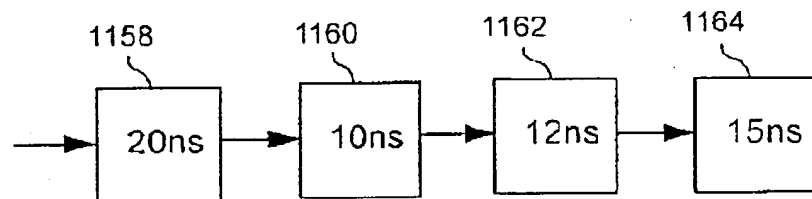


Fig. 81

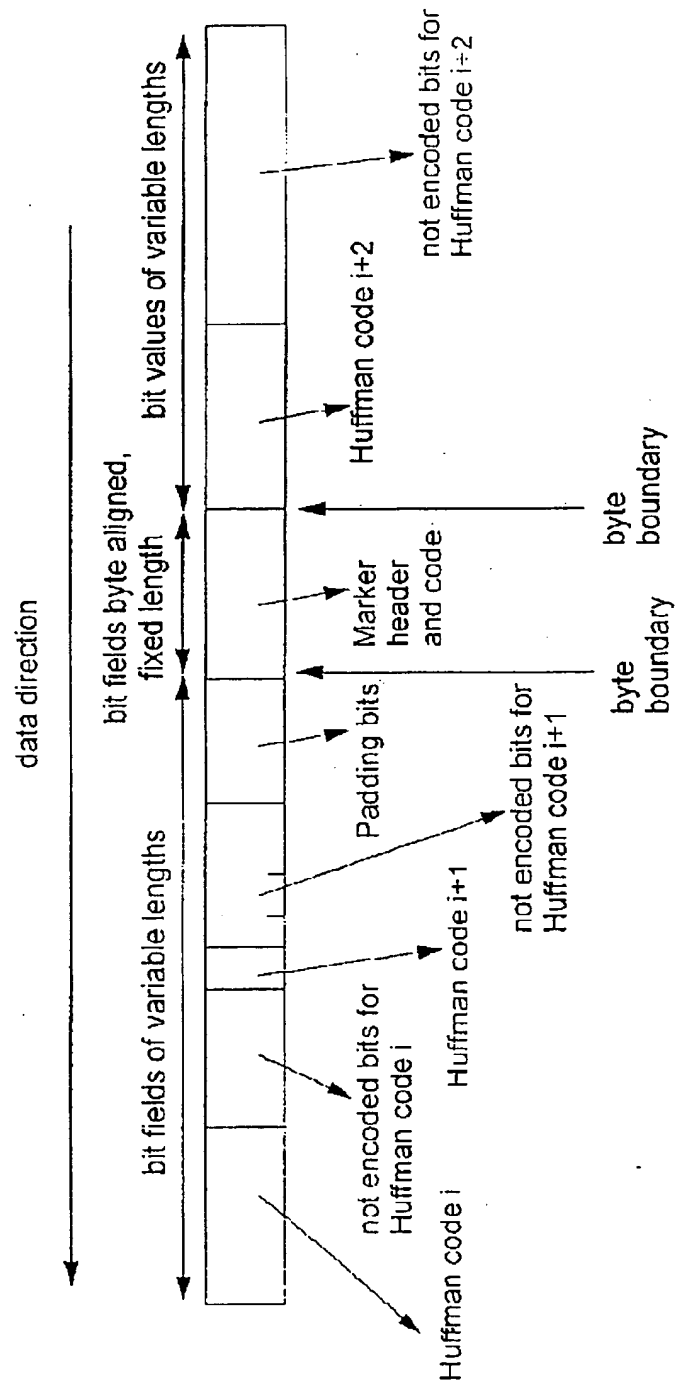


Fig. 82

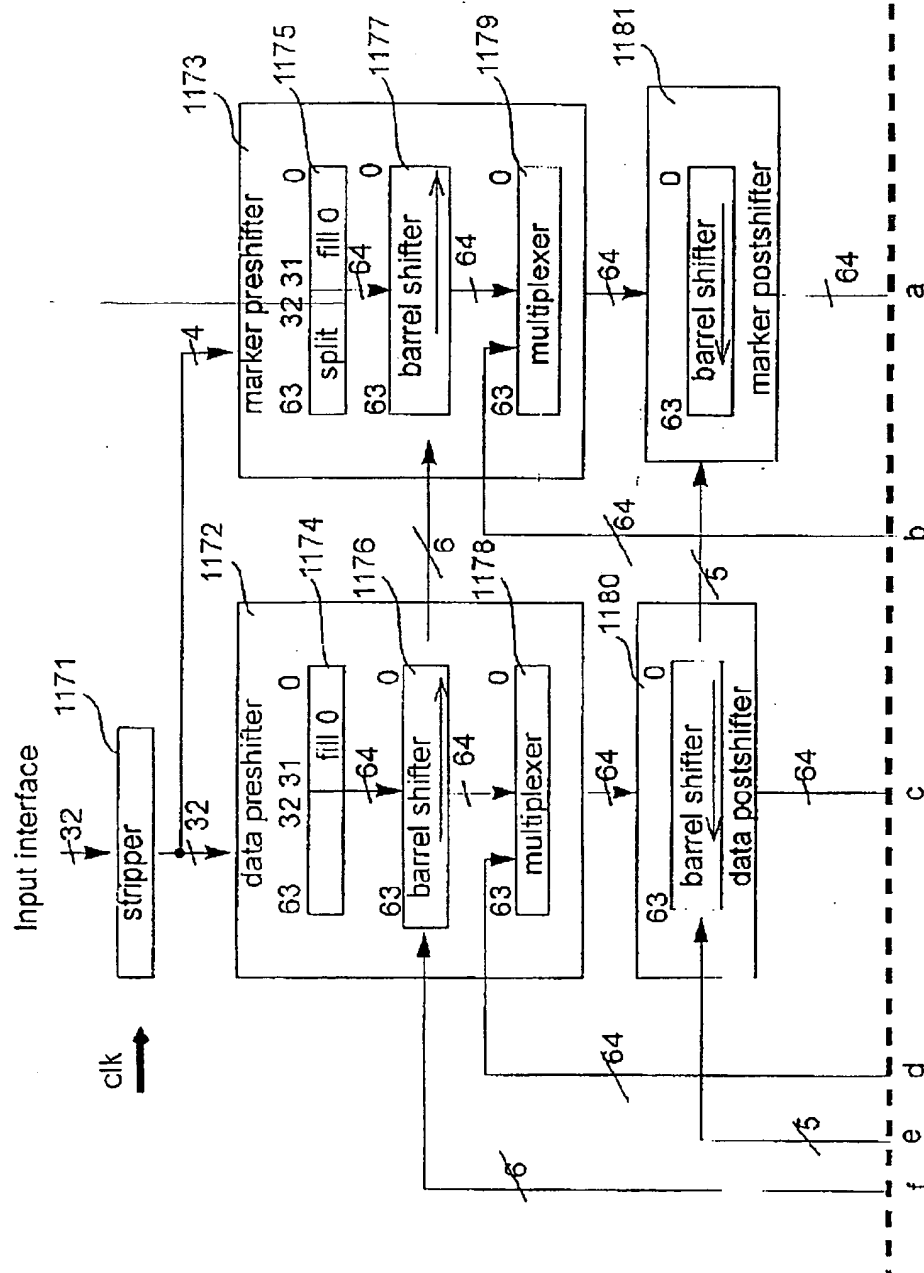


Fig. 83A

Fig. 83A
Fig. 83B

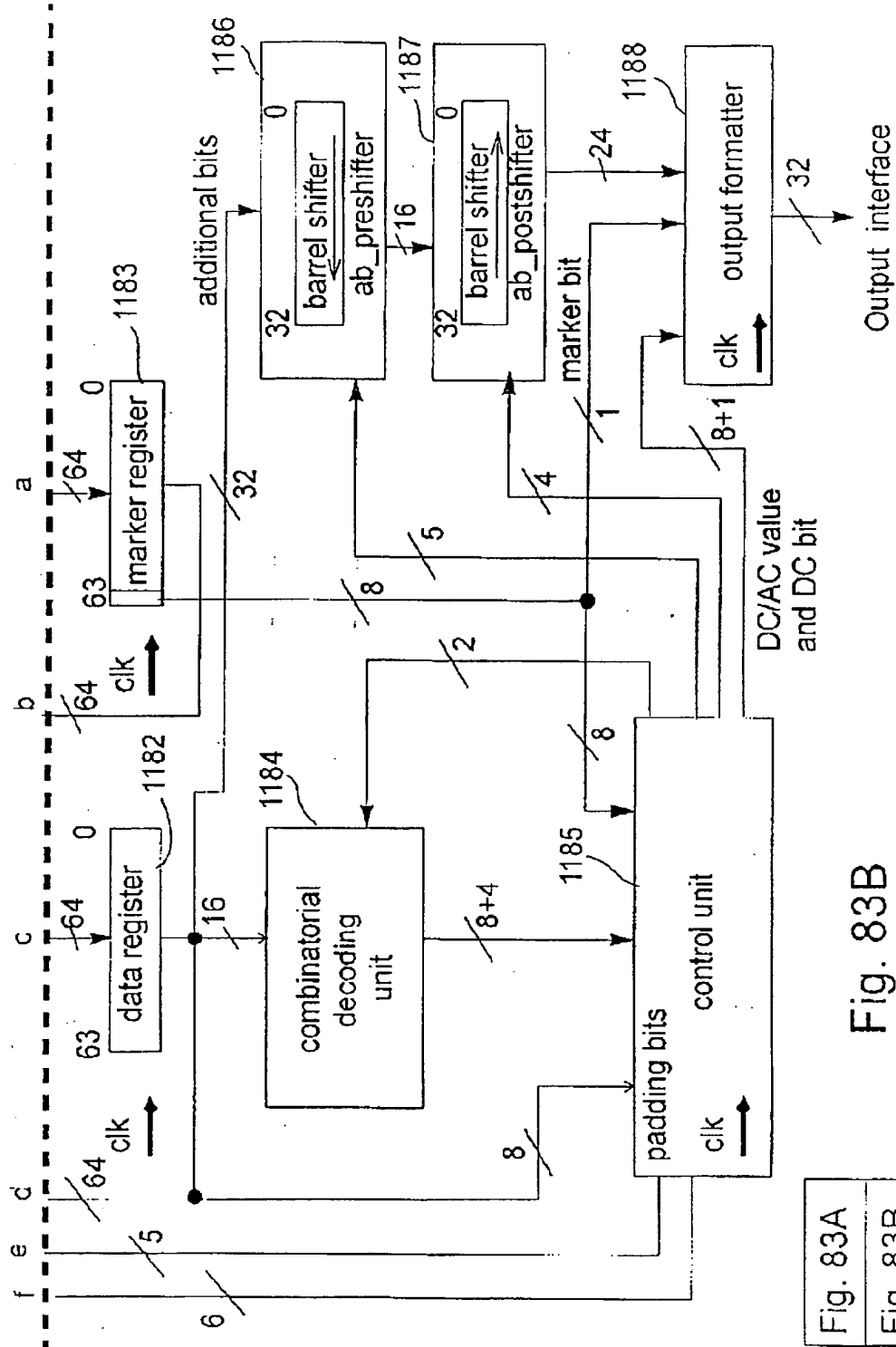


Fig. 83B

Fig. 83A

Fig. 83B

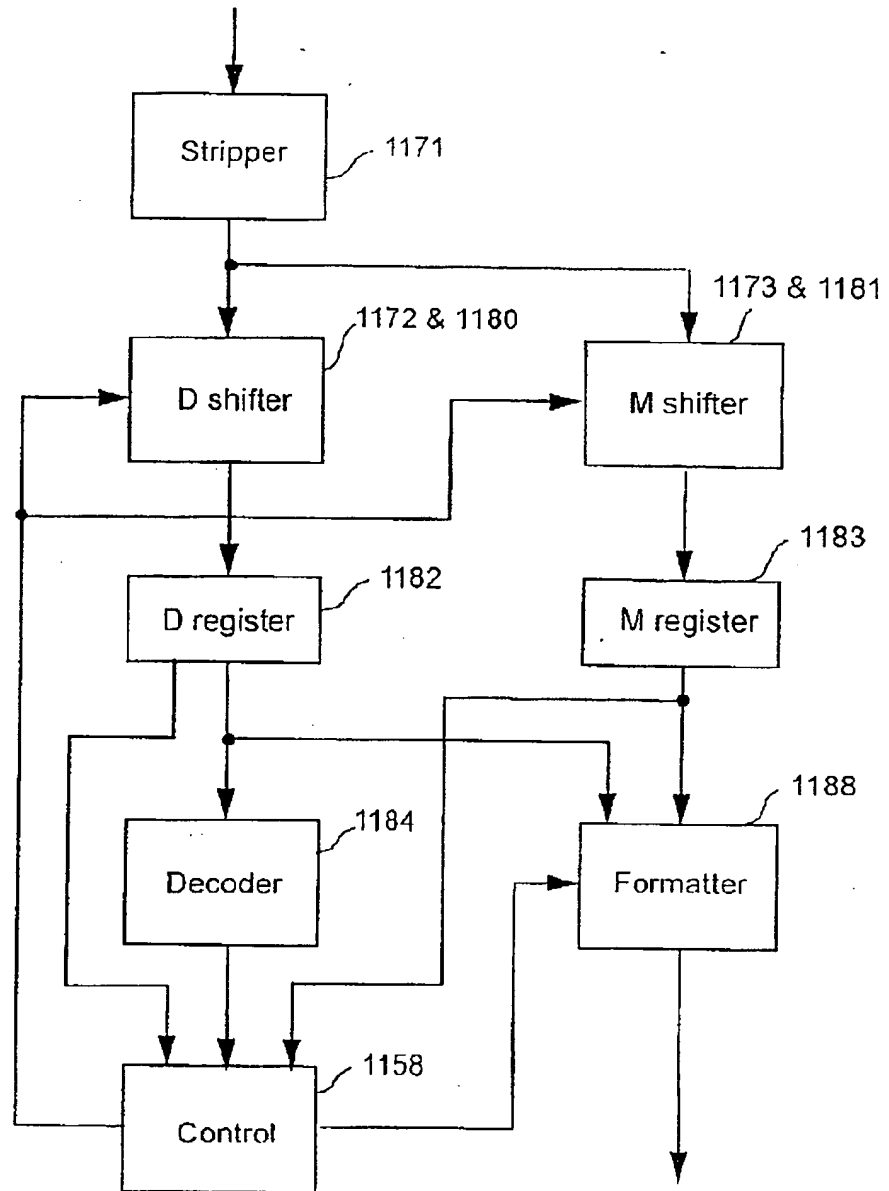
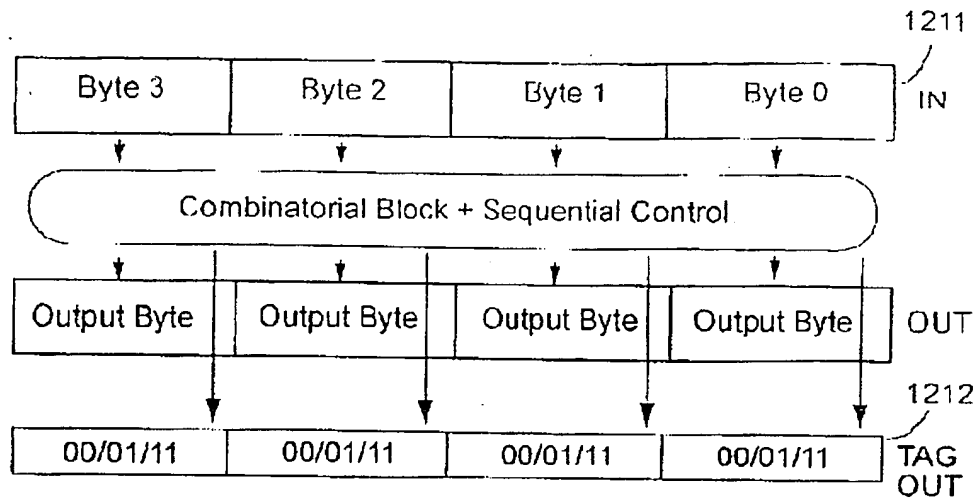


Fig. 84



00 - Stripped Byte Invalid, 01-Valid, 11-Valid and following a marker

Examples of Output Data Produced for a Given Input

1213

INPUT	CORRESPONDING OUTPUT	OUTPUT TAGS
bi - byte to be passed on si - byte to be removed mi - marker byte	bi - byte passed on X - don't care byte	
b3 b2 b1 b0	b3 b2 b1 b0	01 01 01 01
b3 b2 b1 s0	b3 b2 b1 X	01 01 01 00
b3 b2 s1 b0	b3 b2 b0 X	01 01 01 00
b3 s2 b1 b0	b3 b1 b0 X	01 01 01 00
s3 b2 b1 b0	b2 b1 b0 X	01 01 01 00
b3 b2 s1 s0	b3 b2 X X	01 01 00 00
b3 s2 b1 s0	b3 b1 X X	01 01 00 00
s3 b2 b1 s0	b2 b1 X X	01 01 00 00
s3 b2 s1 b0	b2 b0 X X	01 01 00 00
s3 s2 b1 b0	b1 b0 X X	01 01 00 00
s3 s2 b1 s0	b1 X X X	01 00 00 00
b3 m2 m1 b0	b3 b0 X X	01 11 00 00

Fig. 85

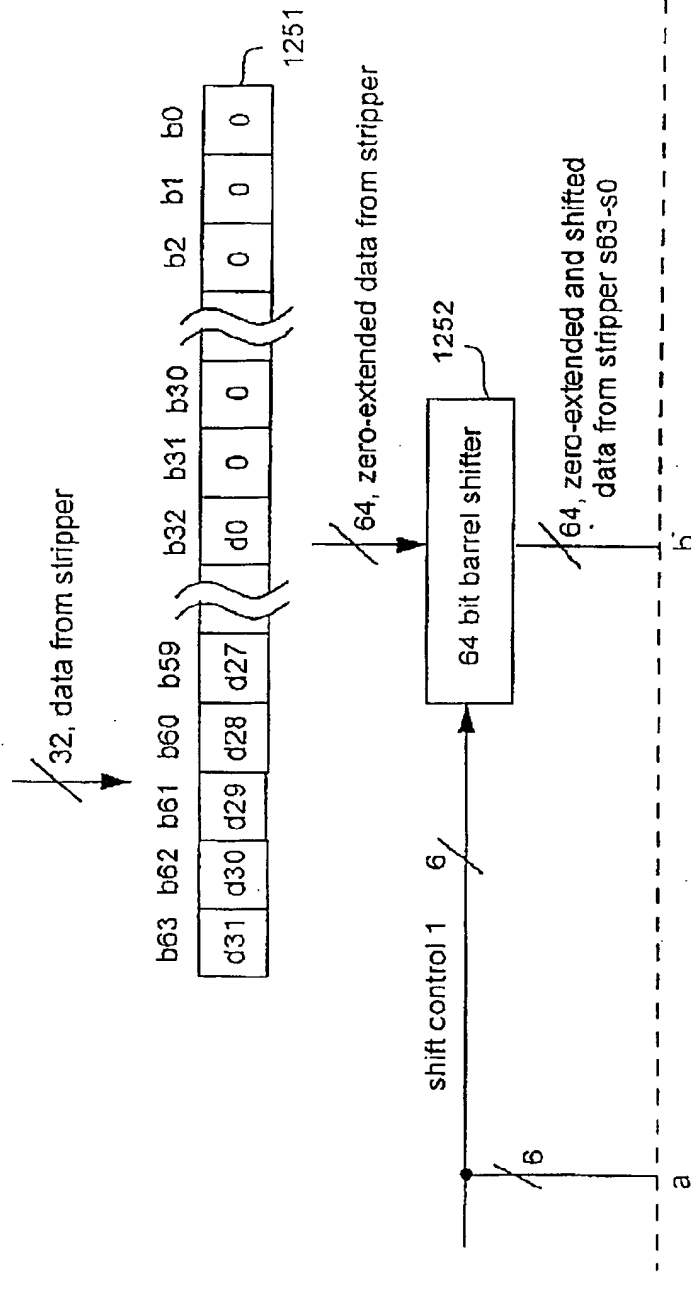


Fig. 86A

Fig. 86A

Fig. 86B

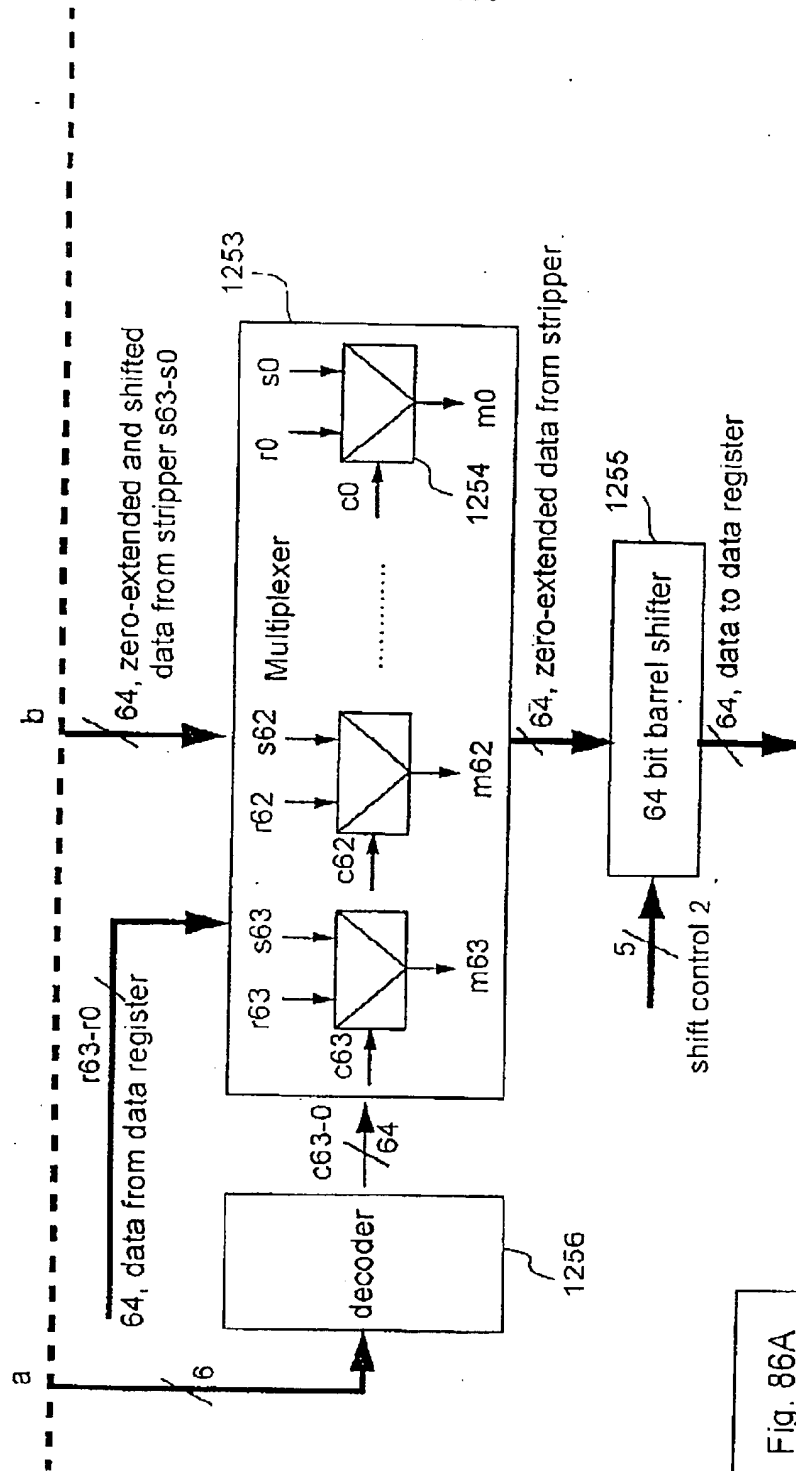


Fig. 86B

Fig. 86A

Fig. 86B

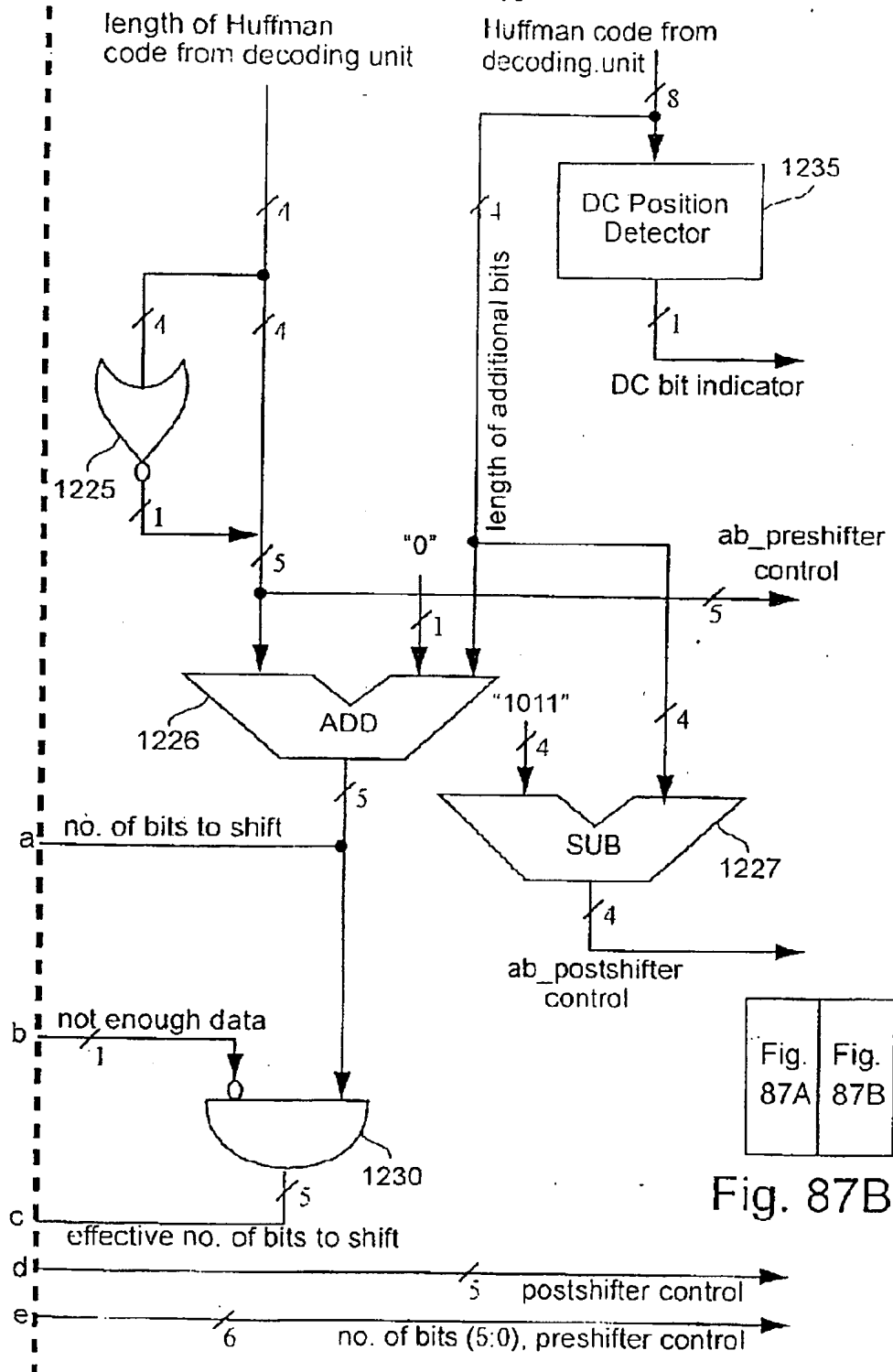


Fig. 87A	Fig. 87B
-------------	-------------

Fig. 87B

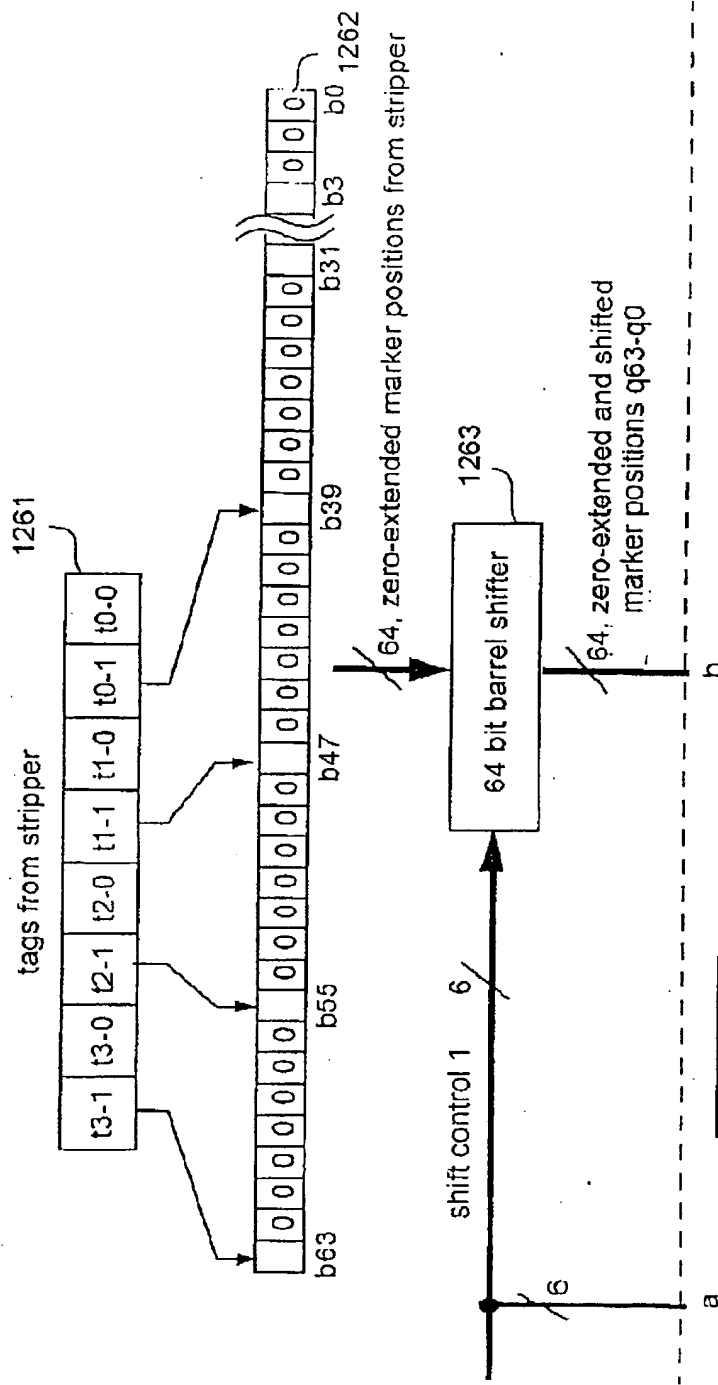


Fig. 88A

Fig. 88A

Fig. 88B

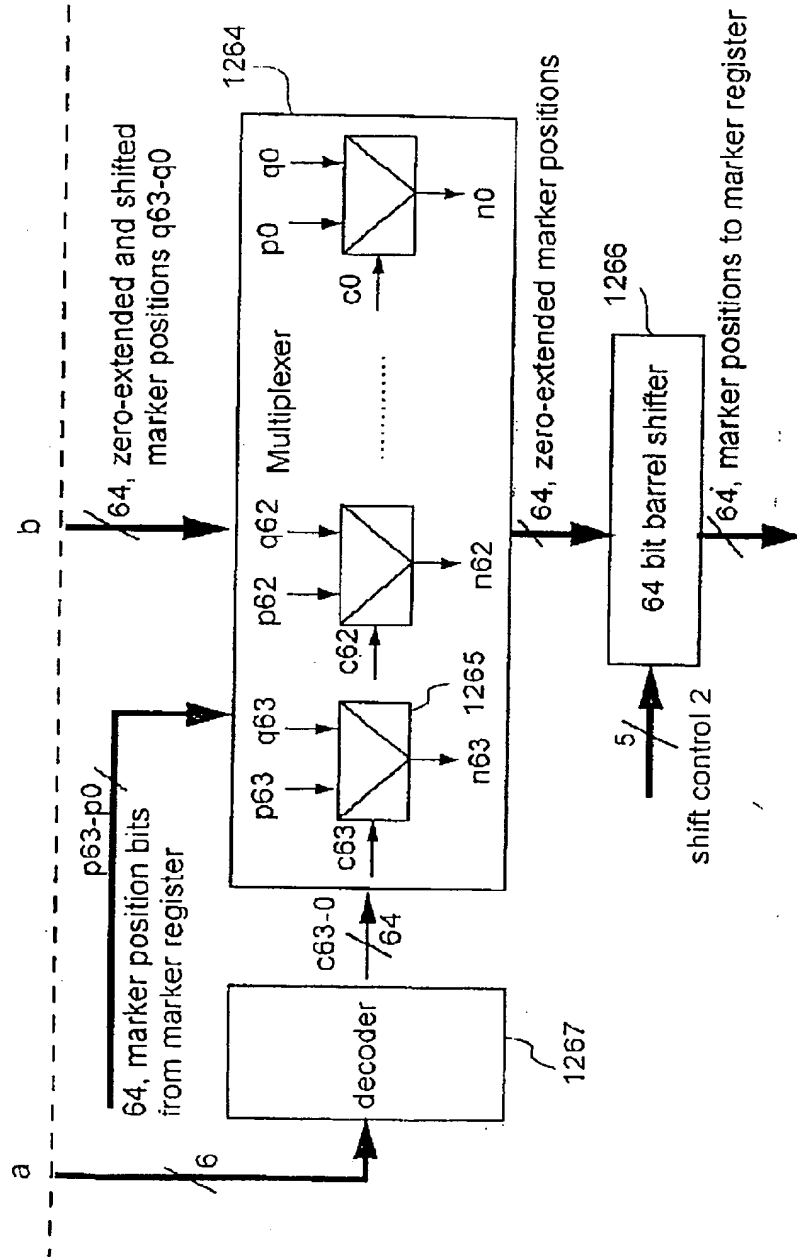


Fig. 88B

Fig. 88A

Fig. 88B